RELAZIONE PROGETTO DI PROGRAMMAZIONE AD OGGETTI

Andrea Bedei, Giacomo Leo Bertuccioli, Lorenzo Gessi, Fabio Notaro

8 Febbraio 2022

Indice

Capitolo 1

Analisi

Il seguente capitolo contiene l'analisi dei requisiti e del problema, ovvero fornisce una descrizione del dominio applicativo e delle funzionalità offerte dall'applicazione.

1.1 Requisiti

Il progetto, commissionato dall'Università di Bologna[1], si pone come obiettivo la realizzazione di un videogioco del genere Tower Defense[2], di nome Siegefend: resist the siege.
Nella categoria Tower Defense, appartenente alla famiglia dei giochi strategici, l'obiettivo è quello di impedire ai nemici di attraversare la mappa posizionando torrette che, sparando automaticamente, bersaglieranno i nemici in avvicinamento.

Requisiti funzionali

- Menù principale all'avvio del software che permetta di iniziare il gioco e modificare alcune impostazioni
- Mappa di gioco, suddivisa in celle di vario tipo (percorso per i nemici e spazio per piazzare torrette), che rappresenta l'ambiente principale
- Gestione dei parametri del giocatore, in particolare vita (che diminuisce ogni volta che un nemico giunge alla fine del percorso) e monete (per l'acquisto di difese)
- Organizzazione del gioco a livelli, ciascuno composto da molteplici ondate di nemici

[1] https://www.unibo.it/it
[2] https://it.wikipedia.org/wiki/Tower_defense

- Creazione di differenti nemici caratterizzati da diversi parametri, tra i quali vita e velocità
- Avanzamento del gioco con difficoltà incrementale (ondate con numero di nemici crescenti e con vita sempre maggiore)
- Piazzamento di torrette che automaticamente sono in grado di rilevare, mirare e sparare ai nemici che entrano nel loro raggio di azione
- Rotazione delle torrette per mirare ai nemici
- Presenza di un negozio in cui acquistare torrette
- Realizzazione e aggiornamento di una classifica basata sul punteggio del giocatore
- Musica di sottofondo
- Classifica di gioco

Requisiti non funzionali

- Fluidità del movimento dei nemici e delle torrette
- Grafica user-friendly e intuitiva
- Sviluppo scalabile ed estendibile del software, in modo da favorire successive modifiche e nuove implementazioni

1.2 Analisi e modello del dominio

Il gioco è strutturato a livelli di difficoltà crescente, ognuno dei quali presenta una diversa mappa di gioco sempre più articolata e una differente organizzazione delle proprie ondate di nemici. Ogni singola ondata è caratterizzata da nemici in numero e parametri differenti.
L'entità nemico dovrà muoversi lungo tutto il percorso della mappa nel tentativo di giungere alla fine dello stesso per togliere una vita al giocatore. Come difesa dall'assalto dei nemici, il giocatore ha la possibilità di posizionare torrette in spazi adibiti della mappa, le quali sono in grado di rilevare e attaccare automaticamente i nemici che si avvicinano sparando loro proiettili. Le diverse torrette sono acquistabili presso un negozio attraverso le monete che il giocatore guadagna eliminando i nemici.
E' inoltre previsto un meccanismo di punteggio del giocatore, il quale si incrementa man mano che i nemici vengono eliminati.
Ulteriori dettagli circa le entità presenti possono essere consultati attraverso la Figura 1.1 inserita di seguito.

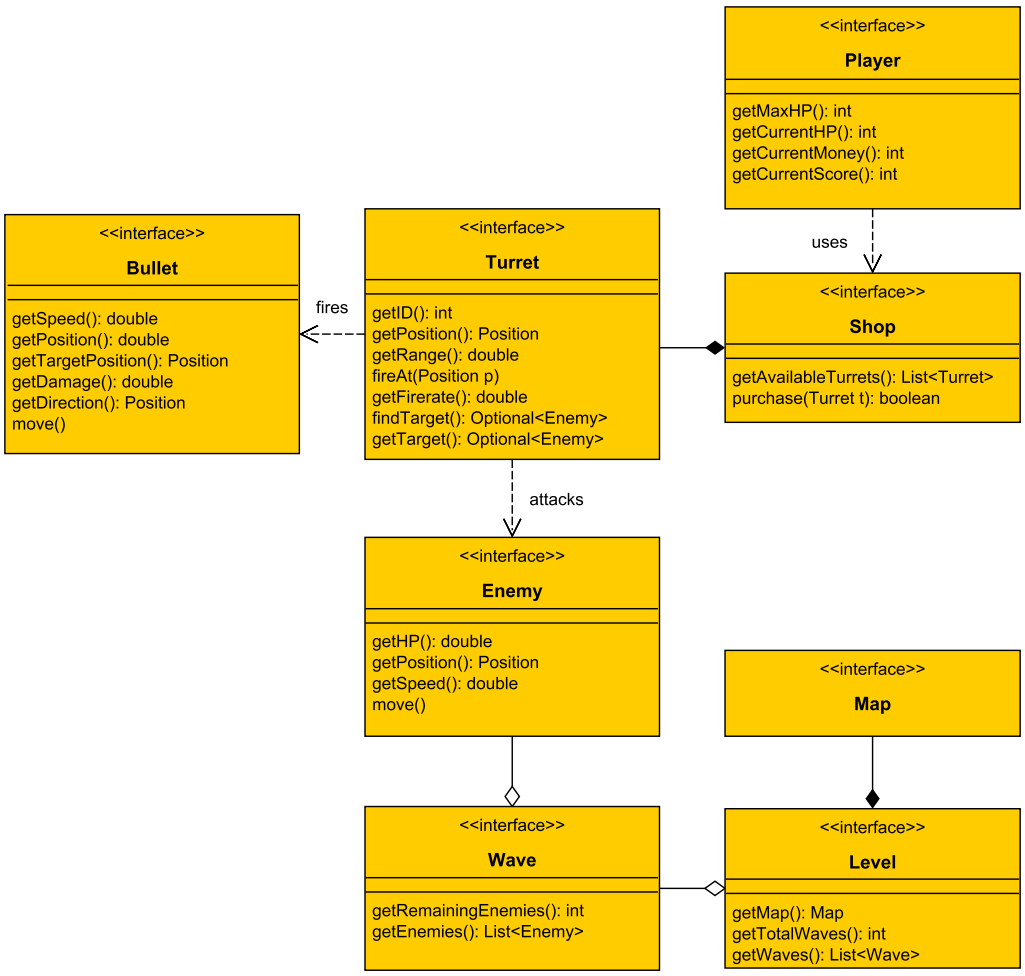


Figura 1.1: Schema UML riassuntivo. Essendo ancora nella fase di analisi, questo potrebbe non essere necessariamente accurato e coerente con il prodotto finale

Capitolo 2

Design

2.1 Architettura

Si è optato per un pattern architetturale derivato dal pattern MVC (Model-View-Controller) che ne è una versione semplificata, in modo da godere di buona parte dei vantaggi che offre pur non avendo buona padronanza dell'originale. In particolare, si è cercato di preservare i benefici offerti dal pattern standard quali l'indipendenza tra parte grafica e logica e la maggiore flessibilità della struttura rispetto alle singole componenti.
La strategia adottata si discosta dal pattern tradizionale per quanto riguarda la mancanza di una singola interfaccia dell'intera applicazione, sostituita da diverse interfacce specifiche per ciascun componente, in modo da beneficiare di un'ulteriore livello di separazione. Tale scelta è stata frutto di lunghe riflessioni circa vantaggi e svantaggi di entrambe le possibilità (ovvero seguire il pattern MVC ordinario oppure la nostra versione).
In particolare è emerso che l'approccio a MVC usuale ha lo svantaggio, oltre alla maggiore complessità di implementazione, di richiedere che ci sia una classe contenente codice relativo ad aspetti anche molto diversi tra loro. Di conseguenza, seguendo questa strada, sarebbero comparse anche classi piuttosto corpose.
La nostra versione, invece, garantisce maggiore suddivisione delle View di gioco, separate in componenti differenti, ciascuno abbinato ad un suo controller, che si coordinano mediante apposite istanze condivise di classi dette Manager. Tuttavia, la nostra strategia comporta lo svantaggio che se fosse necessario cambiare la View, ciò comporterebbe l'implementazione di multiple interfacce (una per ogni componente).

Per meglio comprendere le variazioni della nostra strategia si riporta di seguito uno schema che evidenzia le relazioni tra le View ed i Controller:

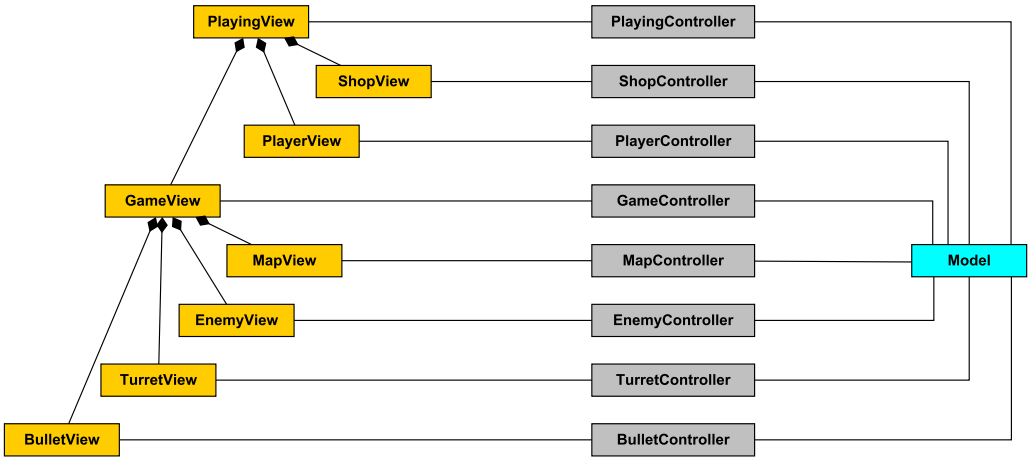


Figura 2.1: Si noti in particolare che l'organizzazione delle view assume una struttura ad albero e ogni singolo componente dispone di un proprio controller, il quale utilizza le classi di Model

Nello specifico il pattern prevede comunque la suddivisione dei compiti in 3 parti:

- Model si occupa di modellare la logica del dominio in classi ed interfacce
- View è responsabile della visualizzazione e dell'interazione con l'utente
- Controller ha il compito di fornire i metodi/funzionalità necessarie alla View per interagire con il Model

Per quanto riguarda l'interazione tra queste 3 componenti:

- il Model incapsula il modello del dominio applicativo del sistema
- la View richiama le funzionalità offerte dal Controller per garantire maggiore suddivisione tra le componenti
- il Controller riceve determinate richieste da parte della View ed aggiorna conseguentemente il Model

Inoltre, abbiamo previsto interfacce e classi che servono a comandare le istanze delle classi di Model, ad esempio EnemyManager, che manipola la classe di modello Enemy così da facilitarne l'utilizzo. Tali classi sono state chiamate Manager, ovvero sono invocate da altri componenti quando ne hanno necessità.

Infine, un'altra categoria di classi sono gli helper, che svolgono compiti specifici e sono usati dai Controller delle View.
Specificatamente al progetto, la struttura generale può essere riassunta nel diagramma che segue:

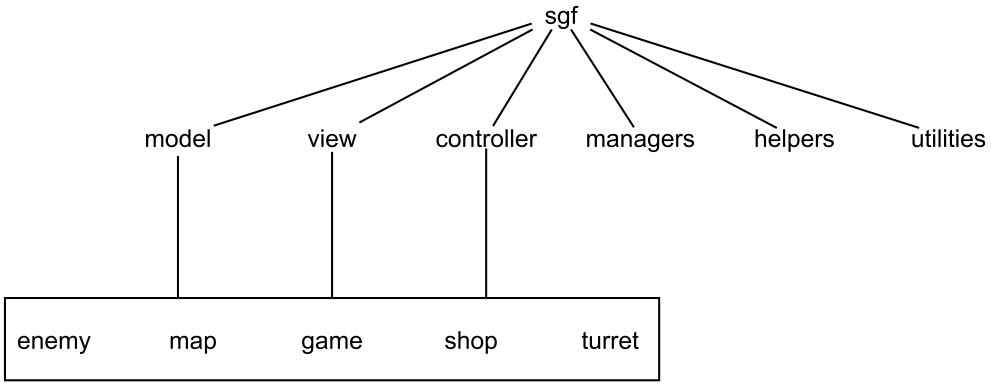
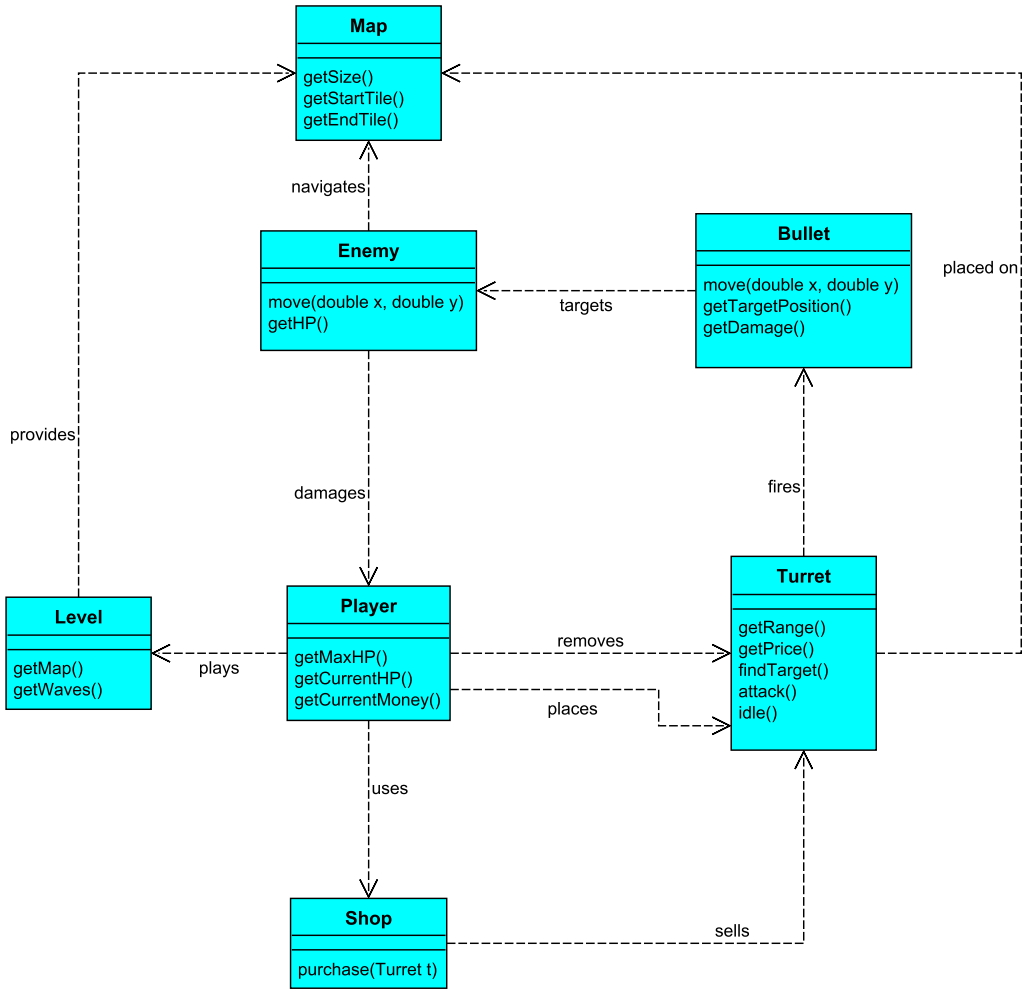


Figura 2.2: Come si può notare, per rendere più pulita la directory del progetto, i package Model, View e Controller contengono al loro interno dei sottopackage ciascuno per ogni parte della view

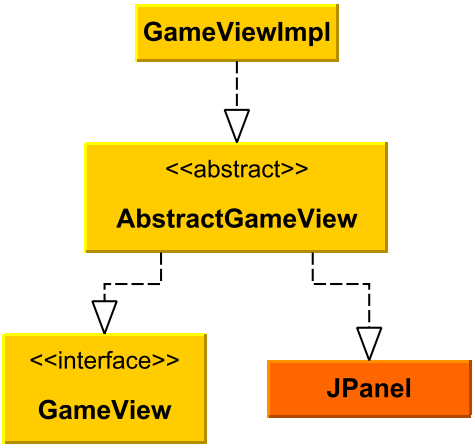
Più approfonditamente, la parte di Model è strutturata come aggregazione di interfacce e classi di dominio (Turret, Shop, Enemy, Map, Player). Ciascuna di queste si occupa di modellare correttamente specifiche entità di gioco.

Le interfacce e le relazioni principali, dal punto di vista Model, possono essere visualizzate nello schema seguente:

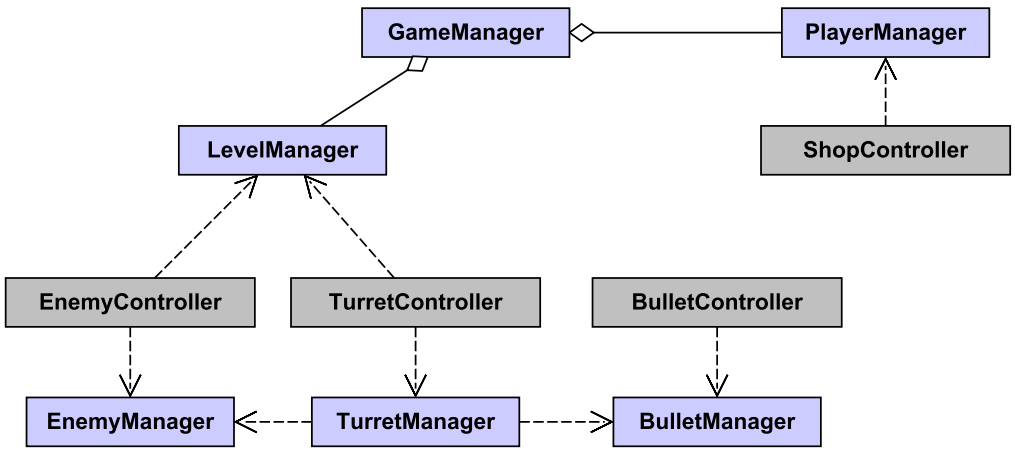


Per quanto riguarda la parte di View, questa è strutturata come composizione di più classi, divise in base a compiti specifici, come il disegno della mappa, la visualizzazione ed il movimento dei nemici, le interazioni con l'utente e molto altro.
Un'ulteriore particolarità riguardante come sono state realizzate le view consiste nell'aggiunta di classi astratte intermedie che estendono JPanel. Questo approccio è stato adottato per facilitare la gestione dell'intera view del gioco (modularità).

Per rendere più chiara questa struttura, si riporta a titolo esemplificativo l'UML nel caso della GameView:



Infine, il Controller è anch'esso suddiviso in classi, ciascuna in grado di comunicare con la rispettiva View e fornire ad essa le funzionalità che richiede. E' importante sottolineare che, a differenza di quanto accade per i Controller, le classi Manager non sanno con chi interagiscono, ovvero non hanno un riferimento ad una particolare istanza di un'altra classe, sia essa un altro qualsiasi componente.
Le relazioni tra i vari Manager possono essere osservate nello schema riportato di seguito:



Un'ultima peculiarità dell'architettura del progetto riguarda la presenza di classi con uno scopo ben preciso e che svolgono lavoro esclusivamente a sup-

porto di altre classi, chiamate helper (MapLoader, WavesLoader...). Si è adottato tale approccio perchè ci permette di spostare parti secondarie di codice in altre classi, in modo da poterle riusare.

2.2 Design dettagliato

BEDEI ANDREA

Il mio lavoro si è concentrato principalmente sulla gestione dei livelli e delle ondate di nemici che li compongono, in particolare ero responsabile della creazione dei nemici e del loro movimento nella mappa di gioco, nonchè delle loro interazioni con le torrette. In aggiunta ho partecipato alla creazione e gestione della mappa, dato che i nemici hanno bisogno di interagire con essa per muoversi nel modo corretto. Infine ho lavorato anche alla gestione della lettura da file, alla creazione della struttura e parti base del gioco, nonchè alla creazione della classifica.

Enemy

Per la realizzazione di Enemy sono state create un'enumerazione (EnemyType) e 3 principali interfacce (nonchè le relative implementazioni): una per ciascuna componente del pattern MVC.
Il primo problema in cui mi sono imbattuto è stato quello di creare un'istanza di nemico e inizialmente avevo pensato di utilizzare il pattern Builder affrontato a lezione. Tuttavia non è stato utilizzato in quanto ci si è resi conto che tali oggetti non erano tanto complessi quanto inizialmente creduto ed inoltre tutti i parametri che gli vengono passati sono obbligatori. Dopo poco tempo è emersa la necessità di creare più nemici, differenziati in base al tipo. Si è pensato perciò di utilizzare il pattern Factory Method, implementando un'interfaccia per la creazione dei nemici e lasciando alla sottoclasse il compito di decidere quale tipo di nemico creare.

Livello

La struttura del livello è cambiata diverse volte durante lo sviluppo del progetto, questo perchè si è deciso di avere un approccio graduale, e una volta impostate tutte le componenti base si è riusciti a creare una specie di gerarchia la quale permette al codice di essere portabile, nel senso che col metodo utilizzato c'è la possibilità di avere un codice facilmente espandibile a più livelli di gioco.
Inizialmente non si era pensato di considerare questo aspetto, siccome il gioco doveva avere un solo ed esclusivo livello, ma vista tale organizzazione è stato facile adattarlo a più livelli.
Ogni livello ha una lista di ondate ed ogni ondata ha dei nemici.

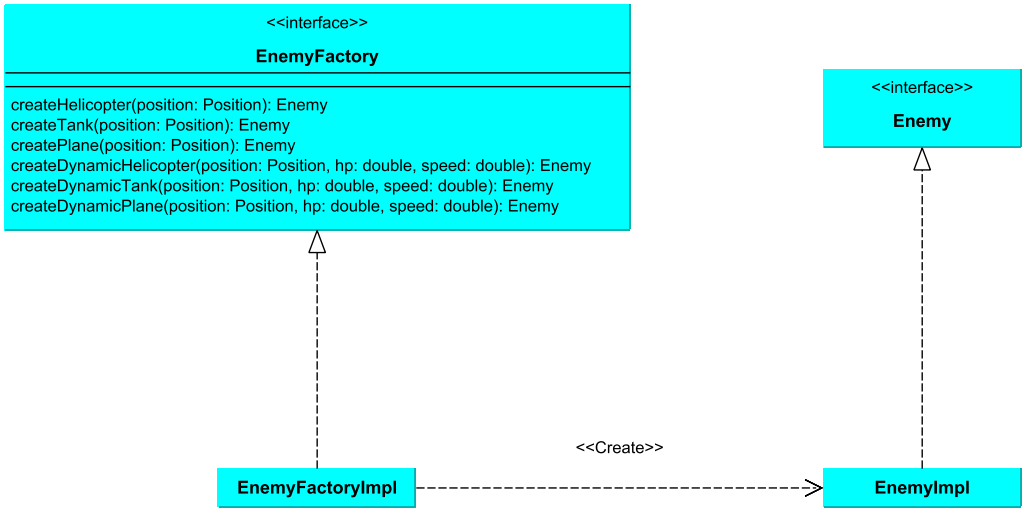


Figura 2.3: Struttura del pattern Factory Method applicato alla creazione dei nemici

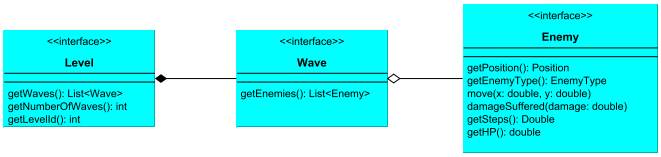


Figura 2.4: Rappresentazione della struttura del gioco (a livello Model) a partire da Level con esclusivamente i metodi più importanti

Gestione nemici

Durante l'esecuzione del gioco, l'EnemyController gestisce la comparsa dei nemici (prendendo i loro dati dal LevelLoader) all'interno del gioco con un determinato intervallo di tempo tra un nemico ed il successivo e tra un'ondata e la seguente.
Inizialmente si era pensato di avere due liste, una per i nemici e una per ogni manager del nemico, ma questo approccio risultava poco efficace nella gestione dell'eliminazione del nemico e nell'interazione con le torrette. Quindi, sempre nel rispetto del pattern MVC il controller crea solo degli EnemyManager, i quali al loro interno hanno il singolo nemico a cui fanno riferimento. Essi permettono il movimento e le interazioni del nemico all'interno della mappa di gioco. Praticamente è presente un thread padre nell' EnemyController che istanzia tanti thread figli ognuno dei quali è proprio un EnemyManager.
Uno degli aspetti più complicati in cui mi sono imbattuto consiste nella rimozione del nemico dalla lista dei nemici attivi nel momento in cui arriva alla fine del percorso oppure quando viene eliminato da una torretta.
Dopo a diversi ragionamenti ho scelto di passare al manager l'interfaccia del controller padre, in modo da eliminare attraverso un opportuno metodo l'EnemyManager preso in considerazione attraverso una notify.
Qui, dopo aver unito le componenti (nemici e torrette), è emerso un importante problema di concorrenza: dalla lista di manager non poteva essere eliminato un nemico mentre questa era utilizzata dalla view, oppure dal metodo di agganciamento delle torrette.
Dopo aver valutato varie alternative (metodi synchronized o utilizzare una copia della lista anzichè l'originale), si è pensato di implementare un semaforo.
In particolare, ho preferito questa strada anche per addentrarmi ulteriormente in un argomento di mio interesse affrontato nel corso di Sistemi Operativi.
Dopo a tale decisione, viene creto un semaforo anche per le torrette e lo sparo.
Inoltre si è dovuto aggiungere un ulteriore semaforo per ogni nemico perchè quando due torrette colpiscono contemporaneamente lo stesso nemico vi deve essere mutua esclusione per decremantargli correttamente la vita.

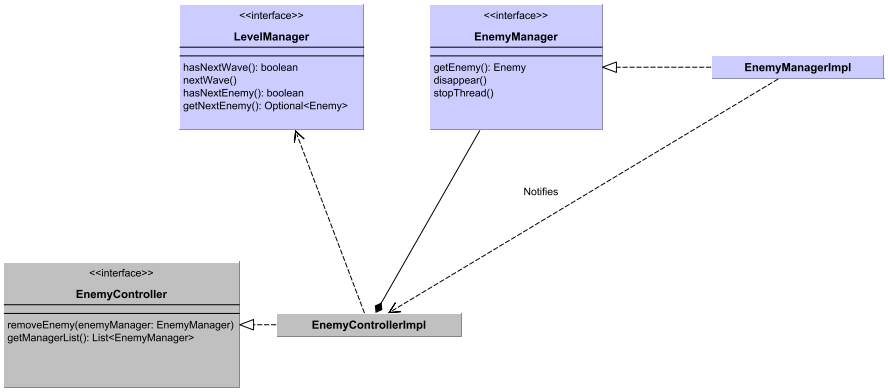


Figura 2.5: Struttura del controller, dei relativi manager e soluzione della problematica dell'eliminazione dei nemici

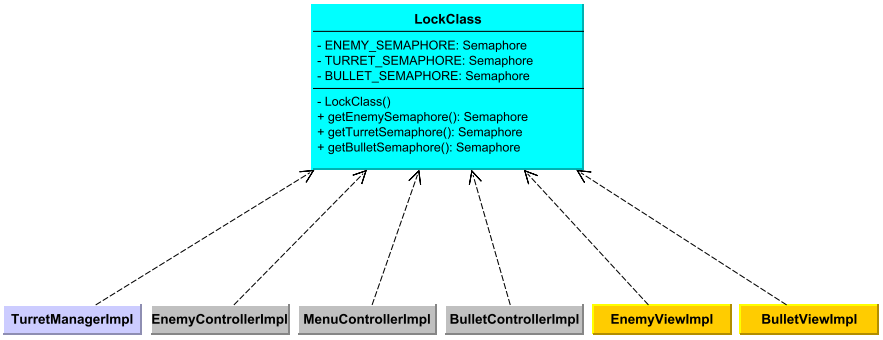


Figura 2.6: Struttura del semaforo e classi dove viene utilizzato

View nemici

Per quanto concerne la View essa lavora esclusivamente con i nemici attivi, i quali vengono disegnati sul pannello riservato ai nemici.
Un aspetto molto complicato è stata la necessità di poter ingrandire e rimpicciolire le immagini dei nemici in base alla dimensione dello schermo, rendendo il codice portabile su diversi dispositivi.
Il controller crea la view relativa ai nemici e gli imposta la lista dei nemici attivi.
La view ha una classe astratta per una motivazione puramente di gestione.

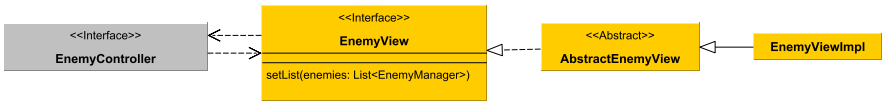


Figura 2.7: Struttura della View dei nemici ed interazione con il Controller

Gestione movimento nemici

Uno degli aspetti più critici incontrati è stato quello relativo al movimento del nemico all'interno delle celle della mappa.
Per affrontare tale ostacolo si è resa indispensabile una stretta collaborazione con lo sviluppatore responsabile della mappa di gioco.
In particolare ciascun nemico compare in una comune e prestabilita cella iniziale fornita dalla mappa per poi muoversi lungo un percorso fino ad una cella finale. Il movimento di ciascun nemico si basa su una direzione, fornita al nemico stesso dalla mappa in base alla posizione che occupa in quel momento.
Più nel dettaglio il nemico capisce in quale direzione muoversi analizzando regolarmente la cella su cui è posizionato, la quale contiene un'indicazione circa la direzione da intraprendere.
Le difficoltà incontrate sono state principalmente: la necessità di convertire posizioni in celle e viceversa, la quale è stata risolta attraverso una classe di utilità; la scelta della frequenza con cui il nemico verifica quale direzione intraprendere (una volta per ogni cella percorsa, in modo da evitare movimenti scorretti all'interno della mappa) e la verifica dello stato del movimento del nemico (capire se il nemico ha raggiunto la fine del percorso, per tutti i possibili percorsi).

Visualizzazione immagini

Un altro aspetto cruciale è stato il caricamento e la visualizzazione delle immagini, metodo implementato in colloborazione col gestore della mappa.
Inizialmente avevamo optato per un caricamento diretto delle immagini, cioè una classe del controller avrebbe caricato tutte le immagini direttamente.

Ma alla fine, cercando di rispettare al meglio l'MVC si è pensato di mantenere in una classe (PathLinkerImpl) i collegamenti tra tipo di nemico e stringa contenente il nome del file immagine da caricare e visualizzare.
Solo allora si è deciso di creare un apposito controller che associa ad ogni tipo di nemico la relativa immagine da mostare in gioco.
Tuttavia si è dimostrato un problema molto simile al caricamento delle immagini di mappa e torrette, quindi, ho progettato e realizzato una strategia in grado di unificare questi aspetti nel rispetto del principio Don't Repeat Yourself: la creazione di un'unica classe contenente tutti i riferimenti tra tipo di entità (nemico, torretta e cella) e nome del file associato.
Per cui, per il caricamento si è pensato di utilizzare il pattern Template Method, permettendo la definizione di uno scheletro generale dell'algoritmo di caricamento immagini, lasciando poi alle sottoclassi specializzate (una per i nemici, una per la mappa e un'altra ancora per le torrette), con uso di tipi generici, l'indicazione degli aspetti più specifici.
E' importante sottolineare che il metodo template è loadRightImage della classe AbstractImageLoader.

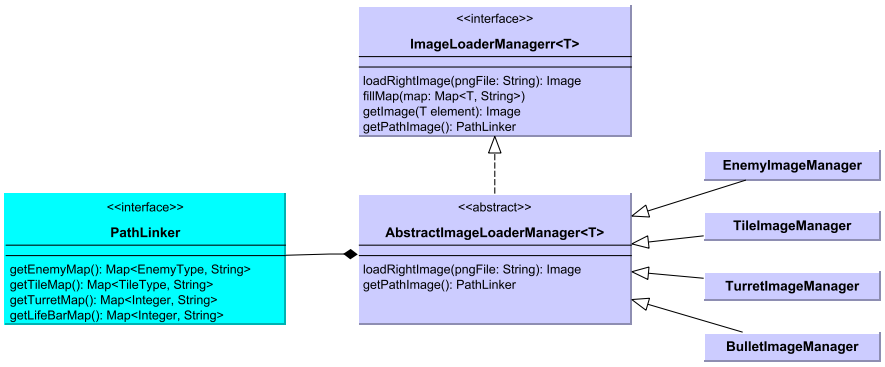


Figura 2.8: Struttura del pattern Template Method per la visualizzazione delle immagini, sia per nemici che per torrette e celle

Creazione classifica

L'ultimo aspetto di cui mi sono occupato riguarda la creazione della classifica. Questo compito ha richiesto una discreta fase di progettazione iniziale, nella quale ho dovuto individuare l'approccio più adatto.
In particolare sono stato indeciso su due approcci da seguire, entrambi riguardanti letture e scritture da file.
La prima idea, forse più facile ma meno adatta, prevedeva di leggere e scrivere in un normale file .txt esattamente come avviene per la mappa e le ondate di nemici.
Questa strada sarebbe stata semplice e coerente con quanto già fatto dai

miei colleghi, ma riflettendo maggiormente sul ruolo della classifica è emerso che per la lettura e scrittura di questa vi sono alcune peculiarità. La principale è che, a differenza della mappa o delle ondate, la classifica è dinamica, ovvero rappresenta uno stato in un preciso istante, invece mappa e ondate sono entità più statiche, definite una volta difficilmente cambieranno.
Dunque ho individuato nell'uso dei JSON la strategia adatta a queste esigenze.
Si è deciso di aggiungere un record alla classifica solamente ogni volta che l'utente completa un livello oppure perde.

NOTARO FABIO

Il lavoro di mia principale responsabilità è stato l'implementazione e visualizzazione della mappa di gioco e la gestione delle musiche, tuttavia ho collaborato attivamente anche alla creazione delle ondate, al movimento dei nemici, aspetto fortemente correlato alla mappa ed all'implementazione dei livelli.

Tile

Fin dall'inizio del progetto è subito emerso il problema di come implementare la mappa di gioco.
L'ipotesi di una mappa "monolitica", ovvero composta da una singola componente, è subito parsa estremamente rigida e poco funzionale, perciò si è scelto di gestire la mappa come matrice (concettuale, non fisica) di Tile, ovvero celle. Questo approccio ha favorito la semplicità e flessibilità del codice, nonchè ha permesso di dividere questa grande entità per poterla gestire separatamente e più facilmente in base al contesto.
Ovvero:

- ad alto livello, ad esempio nella visualizzazione, la mappa viene considerata come un'entità unica
- a basso livello, ad esempio nell'interazione con i nemici o le torrette, essa viene invece considerata come un insieme ordinato ed organizzato di celle.

Le celle possono quindi essere considerate l'unità fondamentale della mappa. Dopo un'attenta analisi sono state individuate le tipologie di celle necessarie (cella di percorso su cui si spostano i nemici, cella di erba su cui posizionare le torrette...), codificate attraverso una enumerazione di nome TileType.
Anche la creazione delle celle è stata sottoposta ad un'approfondita analisi: inizialmente si è pensato di realizzare le diverse celle attraverso il pattern Factory Method. Questo pattern risulta particolarmente efficace per la creazione di famiglie non banali di oggetti attraverso lo scorporamento della costruzione della classe default dalle sue specializzazioni.
Tuttavia, nel caso della creazione di Tile, a variare tra una cella e un'altra

sono solo piccole proprietà (come il tipo di cella), mentre il comportamento rimane identico. Inoltre la creazione di tutte le celle avviene contestualmente e contemporaneamente, per cui si è deciso di rispettare il principio KISS e non sfruttare tale pattern.
Per quanto concerne la gestione delle immagini delle diverse celle, si è deciso di delegare tale compito ad una classe di supporto, chiamata TileImageManager, in modo da non appesantire la classe Tile e rispettare la separazione tra le componenti imposta dal pattern MVC.
In particolare, il compito del TileImageManager è quello di gestire i collegamenti tra TileType e corrispondente file immagine, ovvero restituire l'immagine corretta dato il tipo di cella.

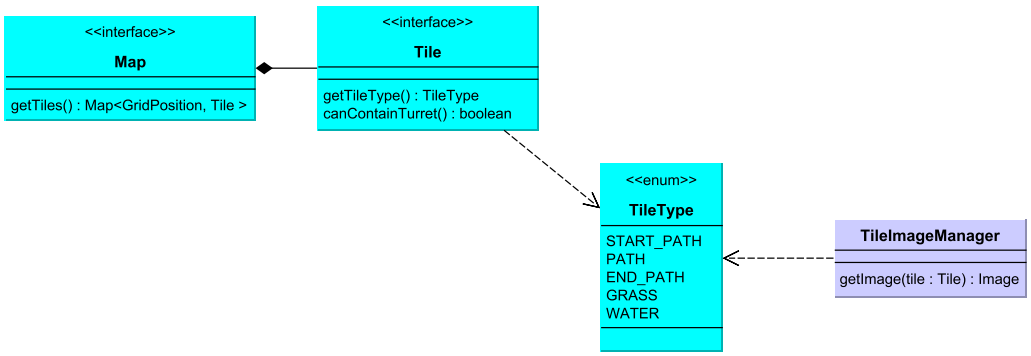


Figura 2.9: Relazioni per la realizzazione del concetto di Tile

Gestione tile sprite

Il caricamento delle immagini operato dalla classe TileImageManager rappresenta in realtà un aspetto implementativo più complesso e per questo è bene trattarlo più approfonditamente.
Inizialmente il caricamento delle immagini delle celle era responsabilità del solo manager TileImageManager che, come detto nel paragrafo precedente, aveva il compito di contenere le relazioni tra tipo di cella e file immagine corrispondente e restituire la corretta immagine dato un tipo specifico di Tile.
Poco dopo, una volta implementato anche il caricamento delle immagini dei nemici, è emerso che il codice per gestire questi due aspetti era pressochè identico in molte parti.
Con l'obiettivo di ridurre le ripetizioni (principio DRY) e scrivere codice flessibile e riusabile, è stato opportuno lavorare in stretta collaborazione con il responsabile dei nemici (Bedei) affinchè si trovasse una strategia in grado di gestire insieme gli aspetti comuni del caricamento delle immagini sia dei nemici che delle celle della mappa(e anche delle torrette), ma permettere al contempo di gestire separatamente eventuali piccole differenze implementa-

tive.
Dopo un'approfondita analisi, la soluzione scelta è stata l'adozione del pattern Template Method. Esso offre infatti importanti vantaggi in una simile situazione, in quanto prevede lo sviluppo della parte comune dell'algoritmo in una classe astratta condivisa, da cui poi ereditano le sottoclassi che si occupano degli aspetti più specifici.
Nel nostro caso, la classe di template si chiama AbstractImageLoader ed essa contiene il metodo template chiamato loadRightImage. Sono poi state realizzate le due classi specifiche, TileImageManager e EnemyImageManager, che gestiscono rispettivamente le associazioni tipo-file delle celle e dei nemici, agendo sì in maniera simile ma con entità del tutto differenti.
Per meglio comprendere la struttura del pattern si consulti lo schema UML fornito di seguito:

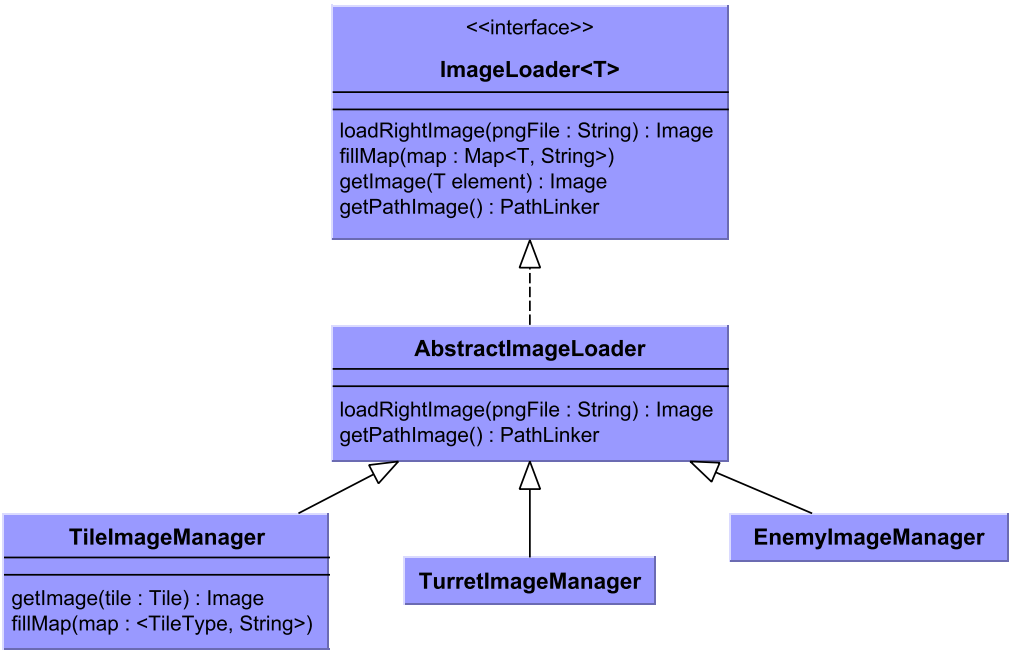


Figura 2.10: Pattern Template Method applicato al caricamento degli sprite sia delle celle che dei nemici

Mappa di gioco

La funzionalità core della mia parte di progetto è stata la realizzazione della mappa di gioco.
Rispettando il pattern MVC è stato necessario suddivdere i 3 aspetti principali della mappa in altrettante classi, una per ogni componente.
La progettazione ed implementazione della parte concettuale della mappa

(intesa come collegamenti posizione-cella) rappresenta sicuramente uno dei problemi più impegnativi affrontati.
Inizialmente la mappa era salvata come variabile di tipo matrice di interi. Leggendo tale matrice ed operando per ogni numero letto una traduzione da numero a tipo di cella veniva creata una seconda matrice con elementi di tipo Tile.
Questa struttura si è dimostrata fin da subito molto rigida ed irragionevolmente poco performante in termini di spazio occupato e risorse per la lettura.
Per garantire maggiore flessibilità al codice e mantenere comunque una struttura relativamente semplice da gestire si è pensato allora di memorizzare su un file di testo la griglia della mappa espressa in numeri, in modo da poter mantenere l'intuitivo approccio precedentemente utilizzato di traduzione da numero a tipo di cella, e leggere quindi da tale file al momento della creazione della mappa. In tal modo non è memorizzata alcuna matrice all'interno del codice (l'approccio precedente ne prevedeva addirittura due), tutto è gestito attraverso una struttura dati di tipo mappa nella classe MapImpl che contiene le corrispondenze tra posizione nella griglia e cella.
Per separare ulteriormente il codice in modo da perseguire una certa modularità ed evitare che il controller (MapController) si trasformasse in una "God class", si è dunque deciso di dividere dal resto (in una classe apposita chiamata MapLoaderImpl) l'aspetto di lettura da file della struttura della mappa da creare e la corretta traduzione di essa in celle.

Il diagramma UML che segue mostra la realizzazione della struttura della mappa di gioco in tutte le sue componenti:

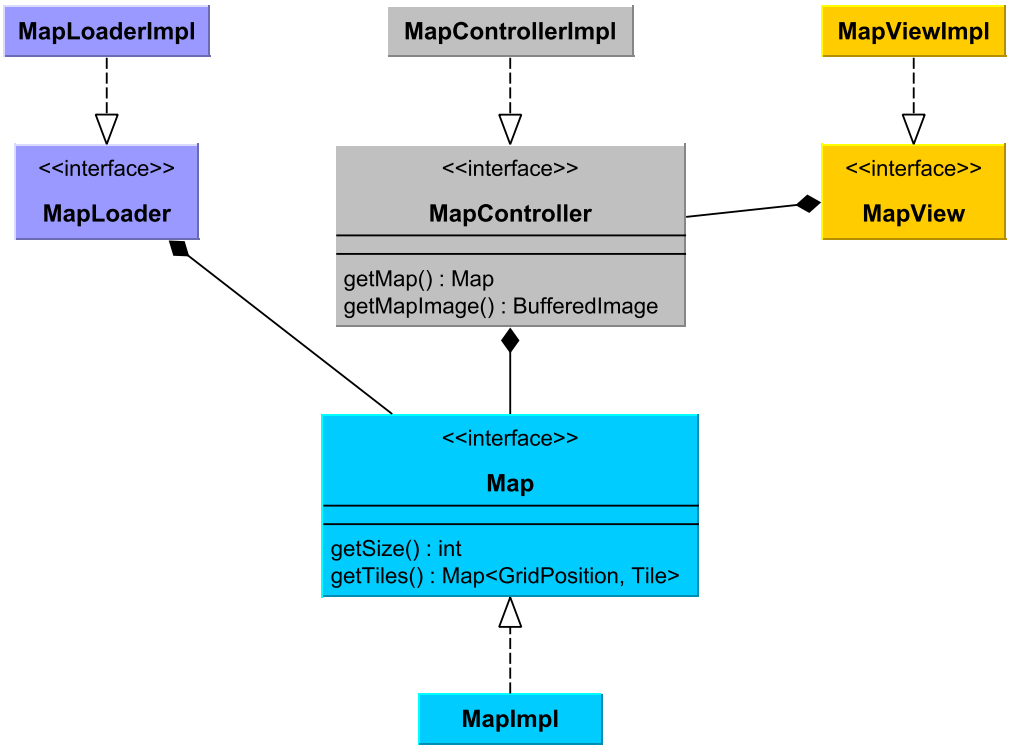


Figura 2.11: Diagramma UML delle componenti create per la realizzazione della mappa di gioco

Per maggiore chiarezza sul caricamento della corretta mappa in base al livello scelto, si consulti il diagramma UML sottostante, che riporta tutte le classi coinvolte dalla scelta del livello nel menu di gioco al caricamento effettuato in MapLoader:

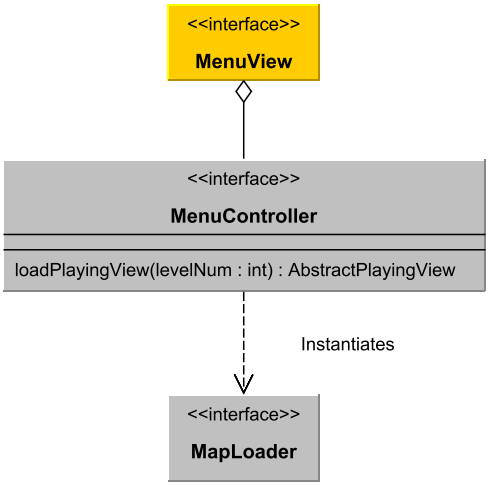


Figura 2.12: MenuView intercetta la pressione del pulsante del livello e richiama il metodo loadPlayingView del MenuController. All'interno di questo metodo verrà dunque creata una nuova istanza di MapLoader, il cui compito sarà appunto caricare la mappa corretta

Caricamento ondate nemici

Ulteriore progettazione è stata richiesta quando è emersa la necessità di decidere in che modo implementare l'avanzamento delle ondate di nemici durante lo svolgimento del gioco.
Fin da subito si è scelto di prediligere la semplicità e coerenza, perciò il caricamento delle varie ondate di nemici è stato realizzato in maniera molto simile al caricamento della mappa di gioco, ovvero esiste una classe (WavesLoaderImpl) responsabile della lettura da file.
Così come per la mappa, è poi stato necessario operare una traduzione dei numeri letti da file nelle varie tipologie di nemici da creare ed inserirli correttamente per formare un'ondata.

Movimento nemici

Altro aspetto che ha richiesto maggiore sforzo di progettazione ed analisi è stato il movimento dei nemici lato mappa.
Si sono ipotizzate molteplici strategie per quanto riguarda il movimento dei nemici.
La soluzione più semplice da implementare prevedeva di leggere da file un'ulteriore matrice che memorizzasse per ciascuna cella la direzione in cui muoversi, tuttavia questa opzione non avrebbe garantito alcuna flessibilità e anzi avrebbe comportato per ogni creazione di una mappa la relativa creazione della matrice di movimenti, cosa inutilmente dispendiosa.
Una soluzione alternativa sarebbe stata fare in modo che il nemico, osser-

vando la mappa e la sua posizione, calcolasse autonomamente la cella su cui muoversi.
Per non far svolgere all'entità nemico questo ulteriore compito, si è allora deciso di gestire comunque in questa maniera il movimento, ma lasciando alla mappa il compito di osservare la propria struttura ed assegnare a ciascuna cella un attributo indicante la direzione del movimento:

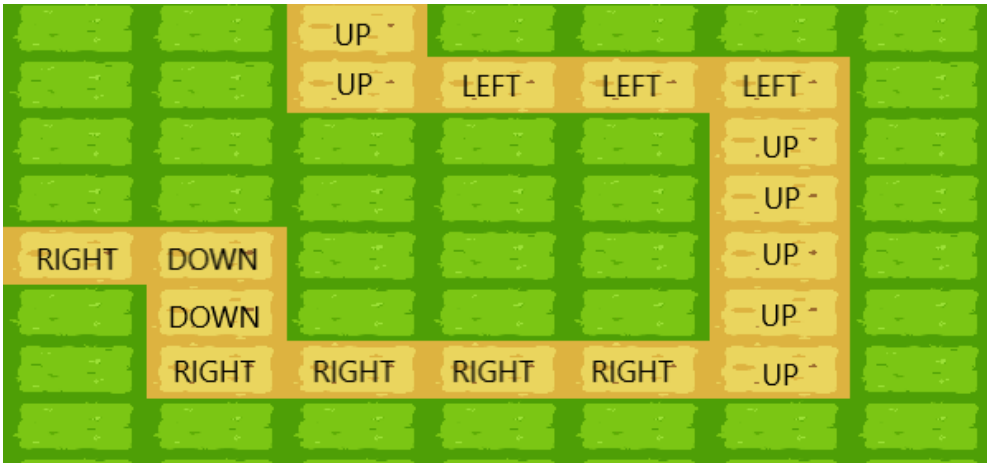


Figura 2.13: Si noti come solo le celle di tipo percorso posseggono l'attributo che specifica la direzione del movimento

La creazione del percorso è dunque realizzata attraverso un semplice algoritmo all'interno della classe MapLoaderImpl che, analizzando una per una le celle che compongono il percorso, riesce ad individuare la sequenza di direzioni da intraprendere per muoversi dalla cella iniziale a quella finale, celle speciali specificate al momento della creazione della mappa stessa.

Sviluppo GUI

Altro punto che ha richiesto un'importante fase di riflessione e progettazione è stato lo sviluppo dell'interfaccia di gioco.
Come espressamente richiesto dalle caratteristiche indispensabili del progetto, la GUI è stata sviluppata con un occhio di riguardo alla flessibilità e adattabilità.
Essa è stata dunque realizzata in modo da analizzare le specifiche tecniche dello schermo su cui è in esecuzione ed adattarsi di conseguenza durante la fase di inizializzazione.
E' stato inoltre ritenuto fondamentale garantire all'utente la possibilità di ridimensionare a proprio piacimento la schermata di gioco.

Terminazione del programma

L'ultimo punto che ha richiesto un grande sforzo di progettazione dettagliata

è stata la gestione dell'interruzione dei thread e delle view.
In particolare mi sembrava più corretto, prima di terminare il gioco a seguito della chiusura della finestra da parte dell'utente o della vittoria/sconfitta, rendere le view invisibili ed interrompere i vari thread (gestiti tramite i loro manager).
Fin da subito ho pensato che implementare il pattern Observer potesse essere una soluzione ideale ed efficace. Questo pattern prevede di stabilire delle dipendenze tra i componenti, in modo che quando uno cambi, gli altri vengano avvisati.
Nel nostro caso ho identificato come observer tutte le classi di view e tutti i manager (per un elenco più dettagliato si consulti il diagramma UML fornito di seguito).
Inizialmente avevo designato la classe GameManagerImpl come observable e vi avevo definito i metodi register (richiamato da tutti i manager e le view per registrarsi e osservare lo stato del GameManager) e stop (che per ogni observer registrato richiamava il suo metodo stop, ovvero interrompeva tutti i thread tramite i manager e rendeva invisibili le view).
Dopo poco tempo, tuttavia, mi sono accorto che utilizzare la classe GameManagerImpl come observable rischiava di compromettere l'architettura adottata, in quanto il costruttore di tutte le view e di tutti i manager doveva averne una copia.
Ho pertanto deciso di creare una classe apposita che funge da observable, chiamata ThreadAndViewObservable, e trasferire qui i due metodi register e stop inizialmente implementati dal GameManagerImpl.
Pur consapevole di non aver adottato un approccio completamente OOP mi è sembrato conveniente rendere statica tale classe, in modo da poterla usare più comodamente dove necessario senza istanziarla (rispetto del principio KISS).
All'atto della chiusura del gioco, dunque, è bastato invocare, esattamente prima della terminazione effettiva del programma, il metodo stop di tale classe ThreadAndViewObservable per interrompere tutti i thread dei manager registrati e rendere invisibili le view registrate.

Di seguito si propone lo schema UML relativo all'implementazione del pattern Observer:

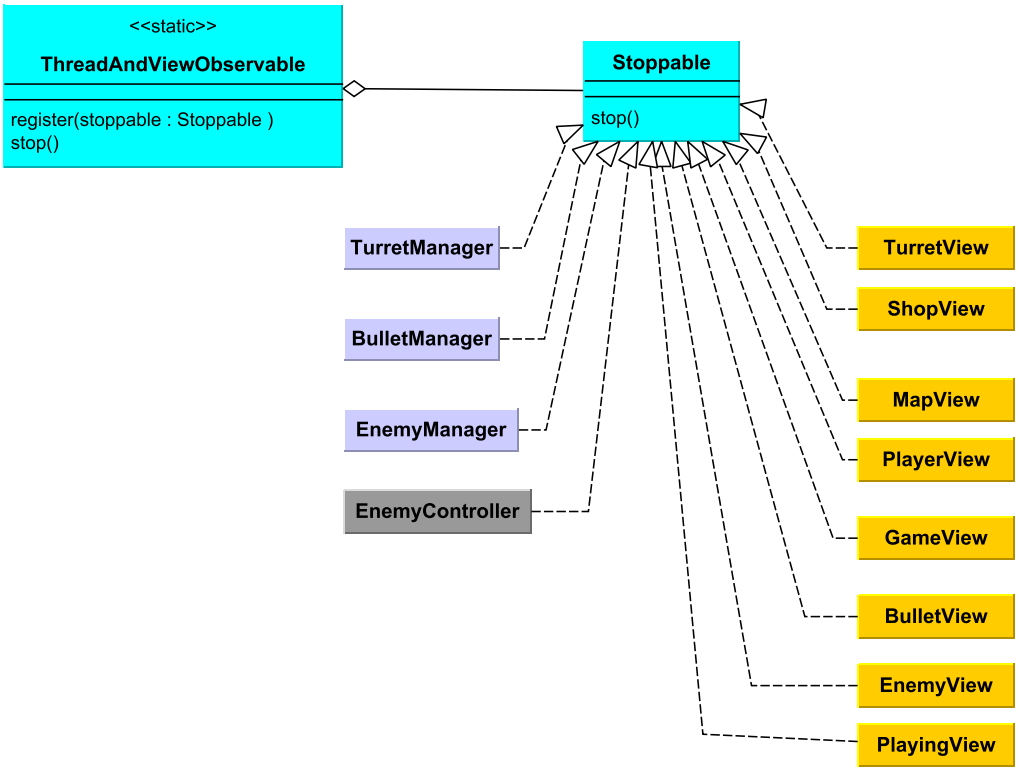


Figura 2.14: Diagramma UML raffigurante l'implementazione del pattern Observer per la terminazione dei thread e delle view

BERTUCCIOLI GIACOMO LEO

Le parti di lavoro da me svolte principalmente riguardavano il negozio e i proiettili nei vari aspetti di view, model e controller. Oltre a quelle, ho realizzato parte delle interfacce necessarie e aiutato gli altri membri del gruppo per risolvere problemi e dubbi da loro riscontrati.

Base per le view e i controller

Tra le prime cose che ho affrontato c'è stata la scelta di specificare una struttura di partenza per view e controller.
Prevedendo un certo tipo di comunicazione tra le due, ho realizzato due interfacce con generici View<C> e Controller<V>, dove i type parameters sono vincolati ad essere implementazioni/estensioni rispettivamente di Controller e View.
Entrambe queste interfacce al loro interno poi contengono il metodo set-

View(..)/setController(..), utilizzato per stabilire un collegamento tra due istanze rispettive di eventuali classi derivate, così da permettere ad esse di comunicare.
In aggiunta, la View contiene anche il metodo start() usato poi dalle implementazioni per evitare chiamate al controller prima che esso sia stato impostato.

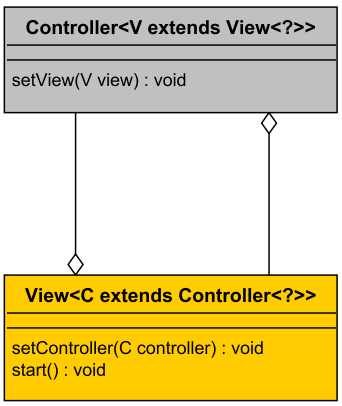


Figura 2.15: Diagramma UML raffiguarnte la struttura di base di view e controller

Shop

Alla base del negozio c'è l'interfaccia Shop, che offre dei metodi per la visione di tutte le torrette disponibili e per controllare se il giocatore dispone di sufficienti risorse per acquistare una torretta specifica.
Shop è implementata dalla classe ShopImpl.

View del negozio

La view del negozio è rappresentata dall'interfaccia ShopView.
La sua implementazione, ShopViewImpl, consiste in un pannello contenente le varie opzioni disponibili affiancate l'una all'altra, ciascuna con un proprio bottone per la selezione.
ShopView è affiancato dall'interfaccia ShopController, che offre alla view dei metodi per ottenere sia la lista di torrette disponibili (usando lei stessa ShopImpl per reperirli in primo luogo) e metodi per selezionare una torretta ed eventualmente anche comprarla.

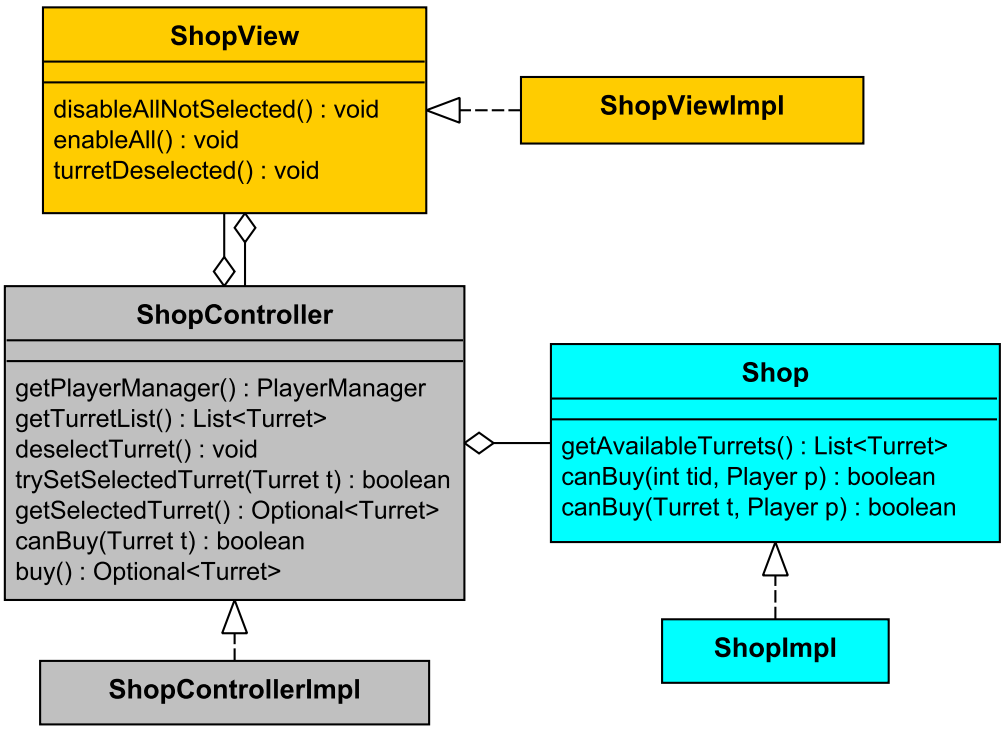


Figura 2.16: Diagramma UML del negozio lato view

Elementi interni del negozio

Nel pannello del negozio, le varie opzioni sono rappresentate tramite sottopannelli contenenti un'immagine della relativa torretta, un'etichetta per il prezzo e un bottone per selezionare/deselezionare la torretta.
Per rendere più pulito il codice, ho trasformato questo sottopannello in una sottoclasse a parte con il nome di ShopItemViewImpl, che estende la classe astratta AbstractShopItemView.
Per la creazione delle istanze di questa classe ero indeciso tra un costruttore e una factory, ma alla fine ho scelto di usare una Static Factory, includendo un metodo statico from() nella classe ShopItemViewImpl.
I sottopannelli così creati offrono metodi utilizzabili dentro ShopViewImpl per cambiare il contenuto del bottone alla sua pressione e per disattivare/riattivare il funzionamento del bottone quando una delle altre opzioni è cliccata.

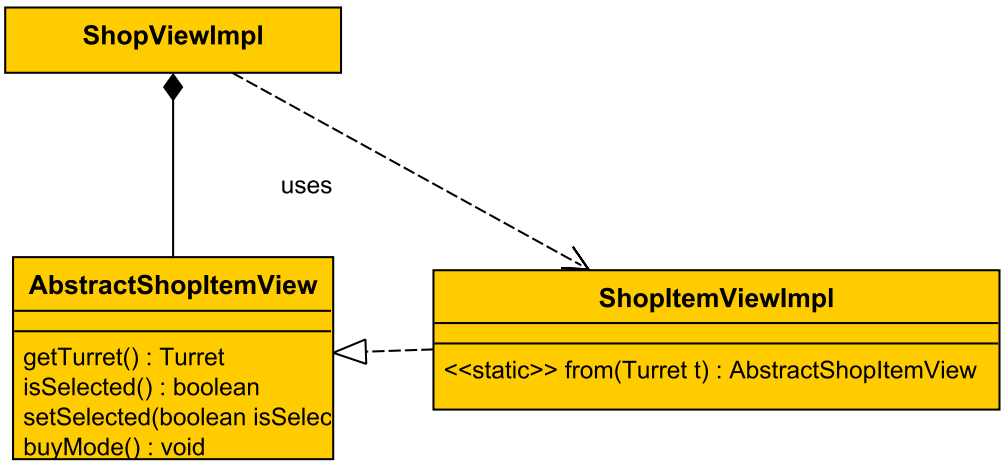


Figura 2.17: Diagramma dei componenti interni della view

Posizionamento delle torrette

Posizionare le torrette richiede un click con il mouse sulla mappa di gioco, ammesso che il giocatore abbia prima selezionato una torretta e che sia in condizione di comprarla.
Al fine di realizzare questa funzionalità, ho dovuto rendere il controller delle torrette in grado di interagire con il controller del negozio, per verificare se una torretta è selezionata e tentare di comprarla.

Movimento proiettili

Come per i nemici e le torrette, il movimento dei proiettili è realizzato facendo uso della classe BulletManagerImpl, con la propria interfaccia BulletManager. Questa classe si occupa anche di verificare che il proiettile sia abbastanza vicino al bersaglio per infliggere danno ad esso e di verificare che il bersaglio esista ancora prima di muoversi verso di esso.

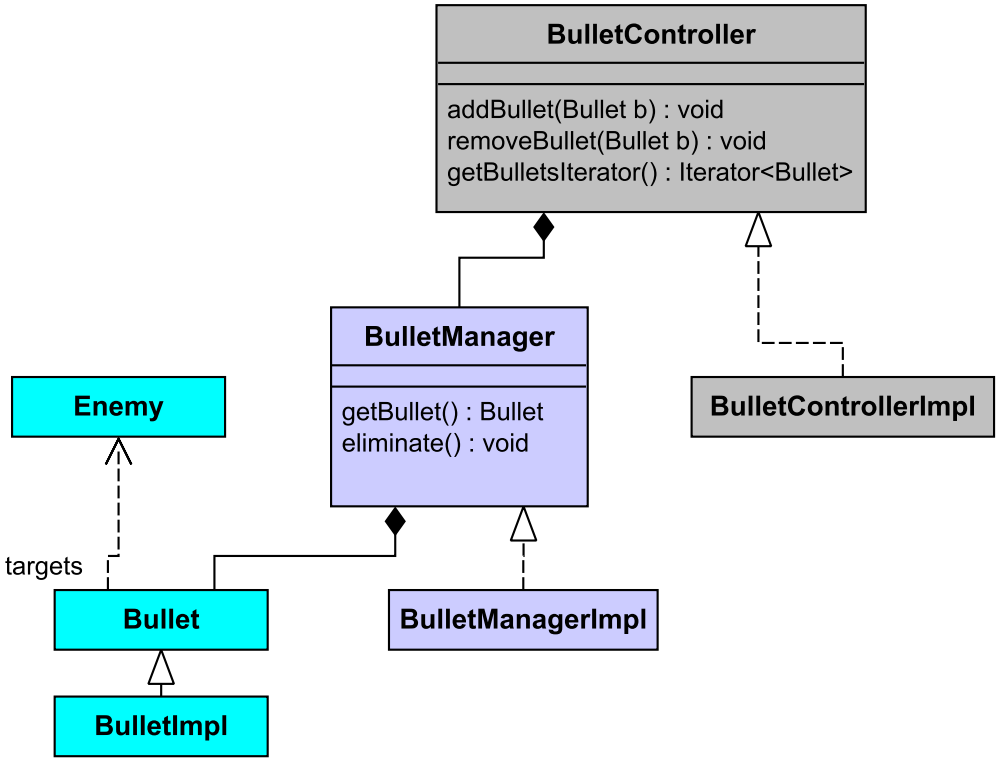


Figura 2.18: Diagramma UML per il movimento dei proiettili

View proiettili

Come per le altre view delle entità attive del gioco, la view dei proiettili consiste in un pannello sul quale è disegnata un'immagine (facendo uso del metodo paintComponent di JPanel) la quale contiene i proiettili attivi nel dato istante.

Mutua esclusione

Siccome sia la view che il controller dei proiettili accedono ad una stessa lista di proiettili con iteratori e Stream, per evitare conflitti si fa uso della classe java.util.concurrent.Semaphore così da garantire mutua esclusione.
Questa strategia è stata adottata anche per approfondire alcuni aspetti del corso visti in precedenza.

GESSI LORENZO

Le mie responsabilità all'interno del progetto si sono suddivise principalmente in 3 sezioni, per le quali mi sono occupato di tutti gli aspetti relativi:

- torrette → al Model, alla View, al Controller e al Manager

- menu → al Model e alla View
- bullet → al Model e alla Factory

In particolare, ritengo che gli aspetti di design più rilevanti per i quali è necessario un approfondimento in questa sezione siano i seguenti.

Menu view

Per quanto riguarda la view del menù ho pensato di realizzarla creando una serie di classi private, ciascuna delle quali rappresenta una sezione del menù stesso; la classe più importante è sicuramente StartMenu, la quale rappresenta l'home screen del nostro gioco.
Essa, attraverso i click dei bottoni presenti, permette di accedere alle altre sezioni del menù.
Sviluppando la GUI, mi sono posto come obiettivo quello di realizzare un'interfaccia:

- user friendly, tant'è che il menù è estremamente straightforward (semplice) anche per l'utente più neofita
- e 'visually pleasing', infatti segue un pattern di colori ben preciso e studiato.

Per rendere il tutto facilmente adattabile e flessibile (a discrezione dell'utente), le classi private che compongono il menu estendono tutte JPanel, così facendo ho potuto creare un layout preciso e pulito.
Al fine di dare omogeneità al look, seguendo il principio DRY ('Don't repeat yourself'), ho deciso di creare una classe chiamata MenuButton che estende JButton nella quale ho impostato il look dei bottoni e aggiunto un ActionListener, così da poter decidere più agevolmente il comportamento dei bottoni.
Infine, per quanto riguarda strettamente la parte di credits, originariamente essa prevedeva una semplice MessageDialog contenente i nominativi dei membri del gruppo, che ho successivamente rimosso, poiché poco gradevole dal punto di vista estetico.
Ho così deciso di inserire al suo posto un testo scorrevole, creando la classe ScrollingText, dando vita ad un effetto molto più piacevole.

Menu controller

Siccome il menù presenta una sezione nella quale è possibile scegliere il livello che si vuole affrontare, ho pensato potesse essere una scelta ragionevole inserire nel controller il metodo loadPlayingView() che si occupa di creare e collegare tutte le entità necessarie e fondamentali per avviare il livello scelto del gioco.

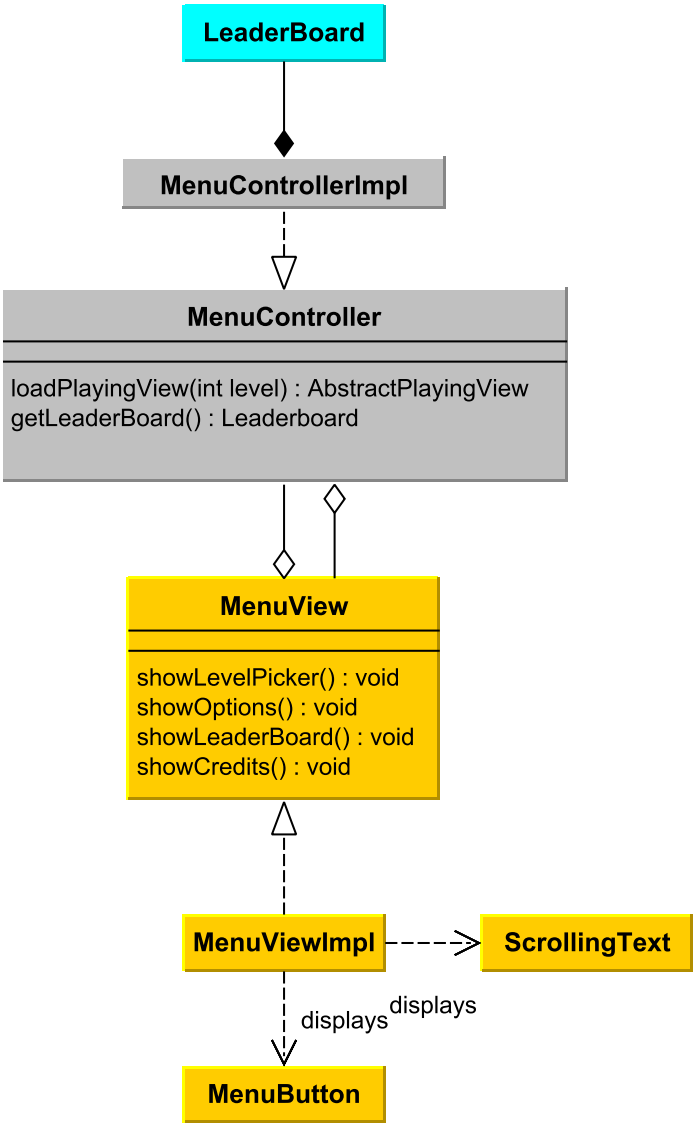


Figura 2.19: Diagramma UML raffigurante View e Controller del menu

Bullet model

Dal momento che i proiettili vengono istanziati dalle torrette, ho valutato che l'opzione migliore fosse, invece di usare direttamente il costruttore della classe proiettile, utilizzare una factory in modo tale da celare l'implementazione del proiettile sottostante, per agevolare la creazione di nuovi tipi di bullets in futuro.

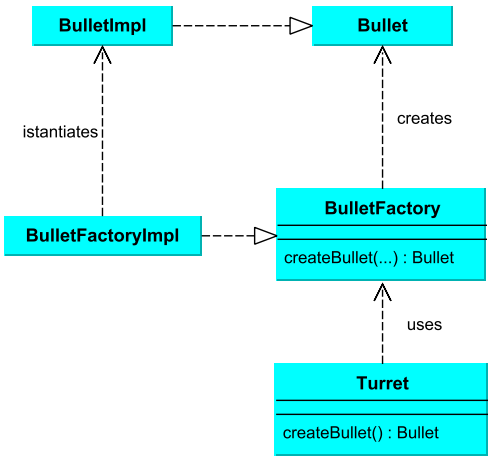


Figura 2.20: Diagramma UML raffigurante la Factory utilizzata per creare proiettili

Turret controller e turret view

All'interno del controller delle torrette è presente il metodo getTurretsIterator() che restituisce un iteratore di torrette cosicché la view possa accedere facilmente a quelle attive.
In alternativa ad un iteratore, avevo preso in considerazione la possibilità di restituire una lista, scelta poi abbandonata perché considerata più dispendiosa e meno 'sicura' (sarebbe infatti stato possibile, per esempio, aggiungere nuove torrette, problema che invece con l'iteratore non si pone).

Turret manager

Per la realizzazione del manager, ho valutato che la strategia migliore fosse quella di utilizzare un thread per trovare un bersaglio e orientare la torretta verso di esso.
Inoltre è servito timer che si occupasse della gestione dello sparo dato che si tratta di un'attività a intervalli, che viene avviato quando la torretta acquisisce un bersaglio e disattivato invece quando la torretta lo perde.
Sintetizzando dunque, ciascuna torretta viene gestita da un manager che si occuperà di tutte le sue attività, quali agganciamento nemico (ovvero trovarlo) e orientamento della torretta stessa verso di esso (attraverso l'aggior-

namento del suo angolo).

Turret model

All'interno del model delle torrette è presente un metodo chiamato getTarget() che si occupa essenzialmente di restituire il bersaglio della suddetta torretta.
La parte interessante però consiste nel suo tipo restituito, ovvero un Optional<Enemy>: chiaramente la scelta di utilizzare un Optional non è casuale, ma è frutto di una riflessione ponderata, poiché ho ritenuto si trattasse di una soluzione più raffinata rispetto all'usare semplicemente il valore 'null' nel caso in cui una torretta non disponga di un bersaglio.

Turret view

Siccome ho previsto che nella view il cannone delle torrette avrebbe seguito il nemico bersagliato, è stato necessario creare un metodo chiamato rotateImage() che si occupasse di restituire un'immagine ruotata della torretta da disegnare poi nella view, sostituendo l'immagine precedente.
Ho deciso dunque di creare tale rotateImage() per scrivere un codice maggiormente pulito e organizzato.

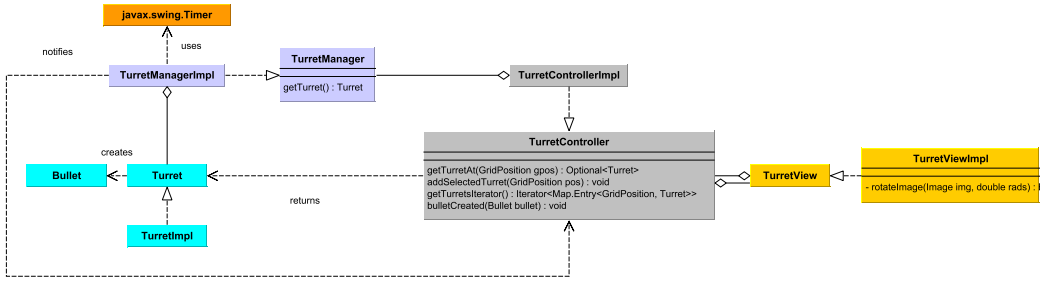


Figura 2.21: Diagramma UML generale del capitolo torrette

VENTURI LUCA (ABBANDONATO)

Questa piccola parte riguarda il lavoro svolto da Venturi Luca, il quale ci ha abbandonato durante la progettazione dell'elaborato, e si tratta di una sezione molto superficiale contenente il lavoro da lui svolto.

- Classe player con gestione del player per quanto riguarda score, vite, monete e username
- Classe PlayerController la quale permette di avere l'istanza player, cambiare gli HP, monete e lo score
- Classe PlayerView permette di vedere una barra con tutte le informazioni sul player durante il gioco

Tali classi sono state create da lui ma abbiamo dovuto fare alcune aggiunte per rendere disponibili tutte le funzionalità del gioco.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

In parallelo allo sviluppo del progetto sono stati effettuati anche diversi test, sia automatici che manuali.
Per quanto riguarda i test automatici, essi sono stati realizzati tramite l'uso della libreria JUnit 5.
Di seguito si riportano le componenti ritenute di maggiore interesse per cui sono stati previsti dei test automatici:

- test sui nemici: si è scelto di sottoporre a semplici test la salute, il movimento ed il corretto caricamento di livelli, nemici ed ondate (compreso il loro susseguirsi durante l'avanzamento del gioco)
- test sulla mappa di gioco: si verifica la corretta lettura della mappa da file attraverso alcuni file di esempio, sia corretti che non. In particolare si verifica la presenza della cella iniziale e finale per il movimento dei nemici
- test sulle celle: si creano alcune celle di prova e si verifica quali possono contenere una torretta e quali no
- test sul negozio: si inizializzano delle torrette, un negozio di base e una view delle torrette per verificare che l'acquisto avvenga correttamente e che i soldi del giocatore siano aggiornati correttamente
- test sulle torrette: sono stati svolti una serie di test automatizzati nella parte del model delle torrette al fine di verificare il corretto funzionamento di alcuni aspetti chiave delle turrets stesse, nello specifico abbiamo controllato che:
  - alle torrette venisse assegnato il prezzo corretto (quest'ultimo letto tramite file.json) - vedi metodo priceTests()

- la posizione iniziale delle torrette, presente nel medesimo file .json, sia assegnata correttamente - vedi metodo initialPositionTests()
- il corretto funzionamento del metodo setPosition() delle torrette, al quale abbiamo fornito come argomento una Position precedentemente creata e poi controllato che la posizione della torretta fosse stata correttamente assegnata - vedi metodo setPositionTest()
- il corretto funzionamento del metodo getClone() delle torrette, per il quale abbiamo invocato il suddetto metodo su una torretta precedentemente creata e in seguito abbiamo appurato che il clone fosse effettivamente uguale alla torretta iniziale - vedi metodo getCloneTest().

Infine, per quanto concerne i test manuali, si è dedicato tempo ad effettuare diverse prove di ridimensionamento della finestra di gioco per verificare che l'utente avesse completa libertà di scelta sulle dimensioni da mantenere.
Il gioco è stato altresì collaudato su diversi sistemi operativi: oltre a Windows 10 e 11, è stata utilizzata una macchina virtuale con sistema opertivo Light Ubuntu 20.04.3.

3.2 Metodologia di lavoro

La parte di lavoro sviluppata in comune tra tutti i membri del gruppo è stata principalmente la parte di analisi iniziale.
Infatti, prima ancora di iniziare a scrivere codice, sono stati necessari diversi meeting attraverso la piattaforma Discord al fine di stabilire e concordare la struttura generale del gioco, ovvero disegnare un prototipo di UML ed elencare le interfacce utili.
In tal modo è stato fin da subito possibile dividere chiaramente il lavoro di ciascun membro del gruppo e procedere con implementazioni autonome.
Sebbene questa fase di analisi iniziale si sia rivelata abbastanza adeguata, è stato inevitabile espanderla tramite l'aggiunta di componenti in seguito alla comparsa di problemi inizialmente non identificati.
Per quanto riguarda il DVCS è stato utilizzato Git, per il quale, in seguito ad una iniziale fase di analisi, si è scelti di adottare una gestione dei branch di tipo "feature branch", ovvero l'implementazione di ciascuna funzionalità del gioco prevedeva un relativo branch, di cui poi veniva effettuato il merge nel branch main una volta terminato il suo scopo.
Nel dettaglio, si è scelto di sviluppare il progetto sui seguenti branch:

- main
- mapImplementation
- enemy

- shop
- enemyMovement
- level
- tests
- music
- bullet
- leaderboard
- menu
- turret
- pause
- player

Invece, per quanto concerne l'integrazione delle parti sviluppate separatamente, i membri interessati hanno creato insieme i giunti, ovvero i collegamenti tra due componenti di gioco, i quali si sono rivelati le parti più critiche della realizzazione dell'intero progetto.
Infine, si riporta di seguito la divisione dei compiti svolti autonomamente.

BEDEI ANDREA
I compiti da me sviluppati sono stati: gestione nemici, gestione struttura, gestione livelli, gestione classifica, movimento nemici, creazione, gestione delle ondate di nemici con l'avanzare del gioco, caricamento dei parametri delle torrette da file e caricamento immagini.

NOTARO FABIO
Le attività da me svolte sono state: gestione celle, realizzazione mappa di gioco, gestione musiche, movimento nemici lato mappa, lettura e caricamento delle ondate di nemici e del livello.

BERTUCCIOLI GIACOMO LEO
Il mio compito principale è stato quello di realizzare il negozio, gestire il movimento dei proiettili, nonché la loro parte di view e controller e gestire il posizionamento delle torrette sulla mappa, ma altre attività che ho svolto sono state:

- creare buona parte delle interfacce di base, poi modificate dagli altri membri a seconda delle necessità;
- realizzare la struttura delle view innestate.

GESSI LORENZO
All'interno del progetto, le macro-attività da me svolte si sono incentrate su 3 elementi portanti del progetto, quali: menu, torrette e proiettili (quest'ultimo compito suddiviso, come successivamente riportato nell'elenco sottostante, fra me e Giacomo Leo Bertuccioli).
In particolare, ho realizzato e implementato la parte di View e Model del menu, il Model, la View, il Controller e il Manager delle torrette e infine il Controller (solo interfaccia), il Model e una classe 'Factory' dei proiettili.
Per chiarezza, si elencano di seguito le interfacce e classi svolte dal sottoscritto (o alle quali ho più attivamente partecipato):

- interfacce:
  - relative a 'bullet'
    - BulletController,
    - Bullet,
    - BulletFactory.
  - relative a 'menu'
    - MenuController,
    - MenuView.
  - relative a 'turret'
    - TurretController,
    - Turret,
    - TurretView.
- classi:
  - relative a 'bullet'
    - BulletFactoryImpl,
    - BulletImpl.
  - relative a 'menu'
    - MenuControllerImpl,
    - AppStart,
    - AbstractMenuView,
    - MenuButton,
    - MenuViewImpl,
    - ScrollingText,
    - ScreenGame.
  - relative a 'turret'
    - TurretControllerImpl,
    - TurretManagerImpl,

    * TurretImpl,
    * AbstractTurretView,
    * TurretViewMpl.

L'elenco precedente riporta, come già specificato, le interfacce e le classi sulle quali ho maggiormente lavorato, ma non include le ulteriori modifiche e aggiunte fatte in alcune delle rimanenti interfacce e classi facenti parte del nostro progetto (nello specifico, Position, GridPosition, MusicController, MusicControllerImpl, PlayerImpl, PlayingViewImpl, LockClass, GameViewImpl ed eventuali ulteriori).

3.3 Note di sviluppo

Di seguito si riportano, per ciascun membro, gli aspetti avanzati del linguaggio utilizzati.

BEDEI ANDREA

Dove possibile ho cercato di adottare le seguenti funzionalità:

- uso di lambda expression e Stream per rendere il codice più compatto e leggibile
- uso di Optional principalmente per quanto riguarda la lista delle varie ondate e dei relativi nemici
- utilizzo di java.util.Semaphore per gestione della concorrenza sulla lista di EnemyManager dei nemici attivi nel gioco, argomento non visto a lezione
- utilizzo di org.json.simple per scrittura e lettura della calssifica su file e caricamento parametri torrette
- utilizzo dei generici per il caricamento delle immagini e utilizzo dei Pair in più modalità.

NOTARO FABIO

Le feature avanzate del linguaggio Java che ho adottato sono state:

- utilizzo dei tipi generici per il caricamento delle immagini
- uso di lamba expression dove possibile per rendere il codice più compatto
- uso di Stream per la lettura da file
- uso di Optional per il campo Direction delle celle (siccome non tutte lo devono possedere)

- utilizzo di javax.sound.sampled per la riproduzione della musica di sottofondo, argomento non affrontato a lezione
- semplice algoritmo di pathfinding (in MapLoaderImpl) per trovare, osservando la mappa, la giusta sequenza di direzioni per andare dalla cella iniziale a quella finale.

BERTUCCIOLI GIACOMO LEO
Nelle mie parti di progetto ho fatto uso di:

- stream e lambda expressions per eseguire azioni sulla lista di pannelli del negozio
- generici nelle interfacce View e Controller
- Optional nella selezione e acquisto di torrette
- Iterator per scorrere attraverso una lista di proiettili
- java.util.concurrent.Semaphore per proteggere iterazioni/aggiunte/rimozioni su lista di proiettili e nemici.

GESSI LORENZO
Tra le funzionalità più avanzate del linguaggio Java, nelle parti di codice da me curate ho utilizzato:

- lambda expressions
- Stream
- Optional
- Iterator
- java.util.concurrent.Semaphore
- java.awt.geom.AffineTransform per la rotazione dell'immagine delle torrette.

Cito di seguito alcune fonti che ho utilizzato:

1. https://bit.ly/3LmDPiQ → per la parte dei credits
2. https://bit.ly/3k5Elpy → per la rotazione dell'immagine delle torrette

Ci tengo a sottolineare però che queste parti di codice sono state prima studiate/comprese, in seguito riadattate agli standard e allo stile del progetto e infine modificate per svolgere correttamente il compito da me richiesto.
Ho approfittato di questi materiali, anche per seguire il consiglio più volte suggerito dai docenti di "non reinventare la ruota da capo".

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

BEDEI ANDREA
Sono soddisfatto del lavoro svolto. In particolare ho trovato molto stimolante l'idea di sviluppare un intero applicativo da zero con i miei amici.
Ho riscontrato molte difficoltà, principalmente inerenti la suddivisione del lavoro e la capacità di relazionarmi con alcuni membri del team.
Credo di aver svolto il mio compito abbastanza bene, cercando sempre di aiutare per qualsiasi cosa avessero bisogno gli altri, tuttavia c'è stata spesso una mancanza di comunicazione e confronto e diverse volte ci sono state discussioni all'interno del gruppo, le quali avrebbero potuto essere evitate attraverso un lavoro più costante di qualche altro membro.
Sono soddisfatto dello sviluppo finale del gioco e di essermi messo alla prova con aspetti con cui non mi ero mai confrontato prima.
Per quanto riguarda il futuro, mi piacerebbe portare questo gioco anche su una piattafroma mobile.

NOTARO FABIO
Mi ritengo molto fortunato ad aver avuto la possibilità di sviluppare un progetto così importante da zero. Sono stato altresì felice della collaborazione offerta dai miei compagni, dai quali ho imparato molto.
Per quanto riguarda gli aspetti positivi della mia autovalutazione, sono innanzitutto molto soddisfatto di essere riuscito a portare a termine quanto mi ero prefissato nella proposta pervenuta ai docenti. Organizzando al meglio i tempi e lavorando con massima dedizione, sono riuscito a rispettare le tempistiche ed a fare anche qualcosa in più rispetto alla mia sola parte obbligatoria.
Dal momento che non mi ritengo una persona troppo creativa, sono stato contento di essermi misurato con problemi che invece mi hanno permesso di lavorare su questo aspetto, come la progettazione e realizzazione della map-

pa di gioco.
Sono stato altresì felice di constatare quanto si siano rivelati utili e diffusi gli argomenti affrontati a lezione, segno che lo scopo didattico del progetto è stato centrato.
Sono stato inoltre piacevolmente colpito dalla profonda disponibilità e collaborazione di alcuni membri del gruppo, che si sono dimostrati sempre propensi ad aiutarmi e a spiegare il lavoro da loro svolto quando non mi risultava chiaro.
Il progetto mi ha tuttavia reso consapevole di alcuni aspetti su cui ancora ho fortemente bisogno di migliorare.
In particolare, ancora non sono propenso ad utilizzare i pattern studiati a lezione quando possibile, nonostante ne abbia compresi vantaggi e potenzialità.
Analogo discorso vale anche per le librerie, che avrei potuto utilizzare per risolvere problemi comuni ma che invece spesso non ho sfruttato.
Ultimo aspetto principale da migliorare riguarda il fatto che ancora impiego molto tempo a comprendere il codice scritto da altri, per cui necessito spesso di leggerlo e studiarlo più volte o richiedere spiegazioni.
In ogni caso, questi aspetti mi hanno permesso di essere consapevole circa i limiti che ancora fatico a superare.
Vorrei infine riportare una riflessione circa la collaborazione con i miei compagni, aspetto rivelatosi molto disomogeneo.
Sebbene con alcuni di questi abbia lavorato molto proficuamente e ci sia una sostanziale ammirazione nei loro confronti, di altri sono stato colpito negativamente in quanto la loro discutibile organizzazione del lavoro ha rischiato di provocare gravi ritardi agli altri membri che invece hanno lavorato sempre con costanza. Questo aspetto ha provocato anche accese disucssioni all'interno del gruppo, cosa che si sarebbe potuta evitare senza troppi problemi se tutti si fossero impegnati al massimo fin da subito.
L'apice di tale situazione si è toccato con l'abbandono di un membro del gruppo, cosa di cui sono estremamente dispiaciuto, anche perchè posso garantire che tutti i membri del gruppo hanno sempre dimostrato massima disponibilità, fiducia e tolleranza nei confronti del membro in questione, cosa che non è stata assolutamente ricambiata in alcun modo. Condanno apertamente il comportamento tenuto dal mio compagno, di cui non condivido neppure le giustificazioni utilizzate con i docenti, le quali risultano non solo immature ma anche false (basti pensare alla stima del tutto irreale del lavoro svolto dal membro in questione, aspetto facilmente confutabile osservando le statistiche offerte da GitHub).
Tutte queste conseguenze non hanno fatto altro che influire negativamente sul morale del gruppo, tuttavia ringrazio profondamente i miei colleghi di non essersi fatti trasportare e distrarre dall'accaduto e di essere riusciti, nonostante questo grande imprevisto, a portare comunque a termine il progetto nella sua totalità e nei tempi previsti.

Altra cosa di cui sono dispiaciuto è stato il fatto che non è mai stato possibile incontrarsi di persona, cosa che ritengo di fondamentale importanza per le fasi iniziali di analisi.
Nonostante questi aspetti negativi non voglio colpevolizzare i miei colleghi, anche perchè ritengo normale il verificarsi di disaccordi e discussioni quando si lavora in grupppo. Volendo è proprio questo l'aspetto più complicato con cui mi sono imbattuto nello sviluppo dell'elaborato.
Infine, per quanto concerne i possibili sviluppi futuri del progetto, mi piacerebbe continuare a sviluppare e migliorare il gioco affinchè un giorno tale prodotto possa essere distribuito a diversi utenti. Più nello specifico, per quanto riguarda la mia parte di lavoro mi piacerebbe apportare modifiche dinamiche alla mappa, in modo che in base a particolari eventi cambi di stile (ad esempio mappa più tenebrosa per Halloween).

BERTUCCIOLI GIACOMO LEO

Lavorare in gruppo è stata tutto sommato un'esperienza positiva, sono riuscito a farmi un'idea di come iniziare a realizzare un progetto e gli accorgimenti da prendere per evitare di dover fare troppe modifiche più avanti nello sviluppo.
Nonostante ciò, in diverse situazioni mi sono trovato frustrato a dover aspettare che altri membri facessero la loro parte di lavoro per poter continuare la mia, soprattutto quando questi interagiscono poco con me (e altri membri), il che mi ha reso complicato il prepararmi in anticipo per finire le parti che mi mancavano.
Riguardo a questo progetto, ho interesse a riprenderlo in futuro, quando ho più tempo, per modificare e sistemare il codice laddove non siamo riusciti a raffinarlo meglio per ragioni di tempo.

GESSI LORENZO

Ritengo che questo progetto sia stato molto valevole per la mia crescita personale, poiché, come primo prodotto universitario svolto in gruppo, mi ha rapidamente posto dinanzi alle innumerevoli novità e sfide che questo tipo di approccio di sviluppo porta con sé, dalla prevedibile difficoltà di doversi correttamente organizzare e suddividere il lavoro con gli altri membri, il saper delegare e fidarsi del lavoro altrui, al sapersi confrontare su questioni per le quali si hanno opinioni contrastanti rispetto agli altri colleghi e il dover accettare che talvolta la propria idea venga scartata in favore di altre proposte e via dicendo.
Non voglio nascondere che avrei gradito cambiare alcuni aspetti del gioco e utilizzare altri strumenti offerti dal linguaggio al posto di quelli effettivamente impiegati, ma alla fine mi ritengo completamente soddisfatto del mio lavoro e di quello dei miei colleghi.
Dato che questa sezione della relazione richiede sincerità e obiettività, cercherò di offrire una panoramica sul mio operato e quello del mio gruppo il

più veritiera possibile.
Personalmente, reputo di aver svolto un ottimo lavoro poiché ho sempre cercato di prestare massima attenzione al modo in cui il mio codice è stato scritto e architettato e proprio per questo ritengo di aver scritto un codice di discreto livello.
Invece, parlando strettamente del mio ruolo all'interno di Siegefend, a causa di questioni strettamente personali e imprevisti non è sempre stato possibile lavorare con la dovuta continuità che avevo originariamente pianificato, nonostante ciò, ritengo di aver sempre cercato di essere disponibile (tramite messaggi, chiamate e via dicendo), malleabile a cambiamenti e spingere alla collaborazione i vari membri.
L'unica critica che mi sento di portare nei confronti di alcuni componenti del gruppo è che talvolta, nonostante io ritenga di aver sempre cercato di cooperare, sono stato escluso da alcune decisioni importanti per il futuro, non solo del progetto, ma anche del gruppo stesso e questo è un aspetto che mi ha infastidito e che vorrei ampiamente evitare in progetti futuri.
Ci tengo però a precisare che, con quest'ultimo paragrafo, non ho intenzione di puntare il dito su nessuno, perché in fondo, nonostante alcune divergenze, sono soddisfatto del lavoro dei miei colleghi che si sono sempre dimostrati disponibili e collaborativi.
Per quanto riguarda il futuro di Siegefend, sono interessato a continuare il progetto, soprattutto per servirmi di esso come un 'playground' per apprendere e sperimentare, difatti all'interno del gioco ho già steso le basi per futuri sviluppi (es. 'upgrades' torrette)

4.2 Difficoltà incontrate e commenti per i docenti

BEDEI ANDREA

Di seguito vorrei condividere alcuni commenti riguardanti il corso.
Innanzitutto ci tengo ad esprimere la mia soddisfazione per come sia stata svolta la prima parte del corso: nonostante gli argomenti iniziali fossero già di mia conoscenza è stato un piacere rispolverarli ed approfondirli ulteriormente.
Per quanto riguarda il progetto, credo sia un'esperienza da fare perchè per la prima volta mi sono ritrovato ad avere specifiche mansioni e a dover collaborare strettamente per giungere alla realizzazione di un prodotto finito, tuttavia non trovo per nulla giusto svolgere questo progetto così impegnativo in ore non appartenenti al corso stesso, portando via tempo prezioso per lo studio delle già tante materie del semestre successivo e per la realizzazione dei progetti che queste richiedono, cosa fondamentale per partecipare agli esami.
Avrei inoltre voluto sfruttare maggiormente le ore di laboratorio per approfondire i concetti affrontati a lezione, invece di passare spesso oltre metà dei

laboratori ad ascoltare slide e spiegazioni su altri argomenti.
Un ulteriore appunto che vorrei far presente è che avrei preferito soffermarmi maggiormente sugli ultimi aspetti visti (Stream, lambda expression e pattern) invece di vederli tutti con estrema velocità solo perchè arrivati a fine corso.
Infine, un ultimo suggerimento riguarda l'esame: rispetto agli anni passati ho trovato equilibrata la parte sui test JUnit, ma non posso dire lo stesso della parte di GUI, in quanto non richiedeva l'uso di particolari implementazioni ma solo logica matematica, aspetto trasversale rispetto agli argomenti del corso.

NOTARO FABIO

Di seguito vorrei condividere alcuni suggerimenti ed approfondire alcune mie riflessioni circa l'organizzazione generale del corso e del progetto.
Innanzitutto ci tengo a sottolineare il mio apprezzamento circa i riferimenti al mondo aziendale e lavorativo che spesso sono emersi dai docenti durante le loro spiegazioni, aspetto non banale. Inoltre ho estremamente apprezzato la completa disponibilità di tutti i docenti a spiegazioni e brevi ripassi quando necessari.
Per quanto riguarda gli aspetti negativi, non ho particolarmente apprezzato l'organizzazione del laboratorio. Avrei preferito approfondire e fare pratica sulle nozioni viste nella parte di teoria invece che affrontare argomenti più che altro incentrati sullo sviluppo del progetto. Alcuni argomenti si sono rivelati comunque estremamente interessanti ed utili (DVCS), invece altri (Gradle) sono stati spiegati secondo me in maniera troppo veloce, a tal punto che non ne ho compreso appieno le potenzialità ed i vantaggi offerti. Inoltre, suggerirei di rivedere il sistema di richiesta di aiuto dei laboratori e semplificarlo.
Infine vorrei condividere qualche riflessione circa il progetto in sè.
Nonostante abbia apprezzato lo svolgimento del progetto e sia fermamente convinto che un'esperienza simile vada fatta, ritengo non sia giusto che un progetto di questa portata e così approfondito venga assegnato nel semestre successivo, togliendo tanto tempo allo studio di altre materie ed alla realizzazione degli elaborati che queste prevedono (già di per sè complicati). In breve, ritengo che le ore dedicate a questo progetto siano in tutto e per tutto ore tolte allo studio e approfondimento delle successive materie.
Avrei dunque preferito svolgere il progetto in parallelo alle lezioni del corso o subito al termine di questo (come fatto per tutti gli altri corsi) in modo da non intaccare il già di per sè pesante semestre successivo ed i relativi elaborati previsti.

GESSI LORENZO

Non ho particolari critiche da rivolgere né al corso né al progetto, ma vorrei semplicemente ringraziare i professori (e tutti i collaboratori) per la completa disponibilità sempre offerta e sottolineare l'utilità dei riferimenti al mondo

del lavoro compiuti dai docenti duranto lo svolgimento del corso.
Per quanto riguarda strettamente il progetto, lo reputo un'attività importante che tutti i miei futuri colleghi dovrebbero poter svolgere.

Appendice A

Guida utente

All'avvio dell'applicazione si presenta la seguente schermata di menu iniziale:

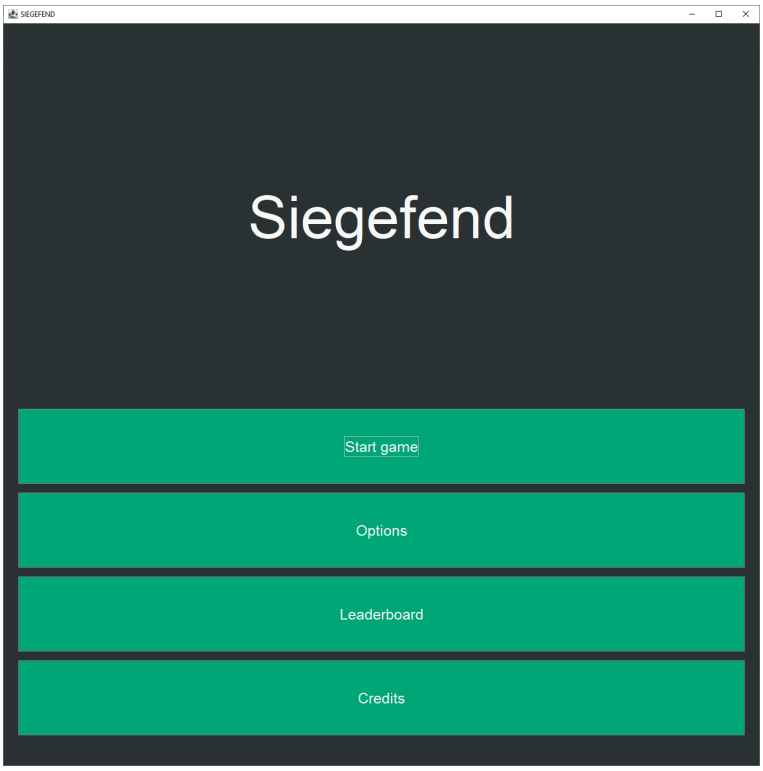


Figura A.1: Schermata di menu iniziale

Prima dell'avvio del gioco, concentriamoci sulle altre azioni proposte:

- premendo il pulsante Options comparirà una nuova schermata in cui è possibile disattivare o riattivare la musica

- premendo il pulsante Leaderboard comparirà una nuova schermata in cui è possibile consultare la classifica ed i punteggi dei vari giocatori
- premendo il pulsante Credits comparirà una nuova schermata con un'animazione contenente i nomi di tutti gli sviluppatori.

Invece, premendo il pulsante Start Game comparirà una schermata cui occorre prestare attenzione:

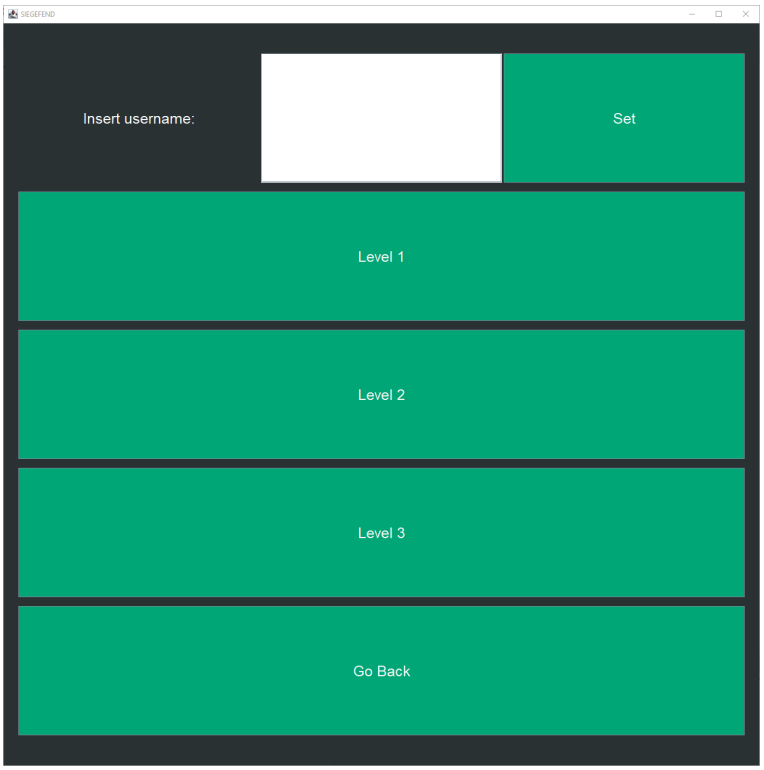


Figura A.2: Schermata relativa al menu di inserimento del nome utente e della scelta del livello

Sebbene siano stati previsti meccanismi di errore o mancato inserimento del nome utente in modo da lasciare all'utente completa libertà, i corretti passaggi da effettuare sono: digitare il nome utente nella text box preposta, premere il pulsante Set per inserirlo e solo dopo premere il pulsante del livello che si vuole caricare e giocare.

Una volta scelto il livello, inizierà il gioco vero e proprio, presentando la seguente schermata di gioco:

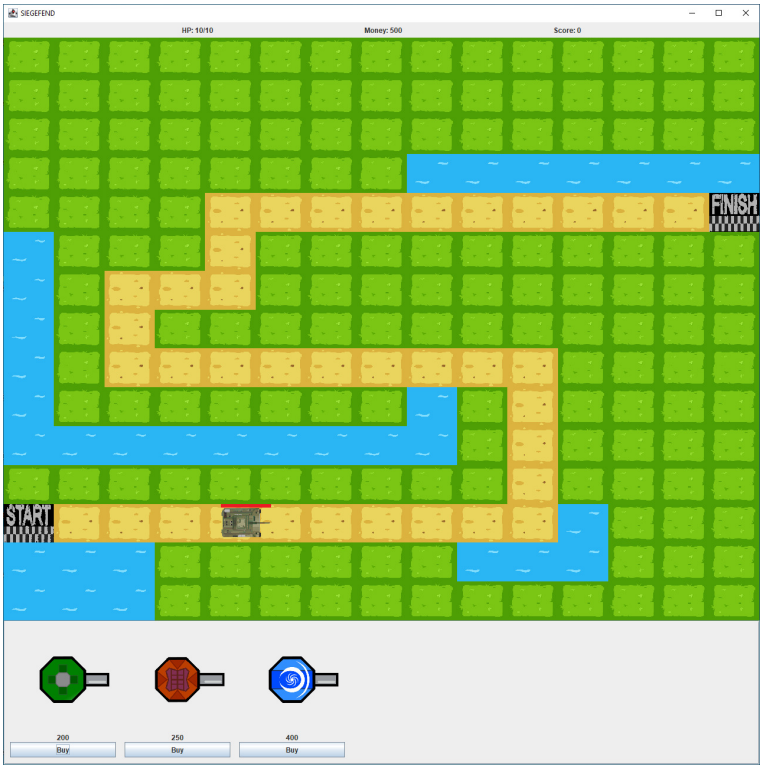


Figura A.3: Schermata di gioco effettiva (mappa di gioco)

I nemici cominceranno immediatamente a muoversi. Per piazzare una torretta basterà premere il pulsante Buy sotto quella desiderata e poi premere la cella di mappa in cui si vuole effettuare il piazzamento (si ricorda che le celle di acqua e di percorso non possono ospitare torrette).

Invece, premendo una torretta già piazzata si potrà vedere il suo range di fuoco:

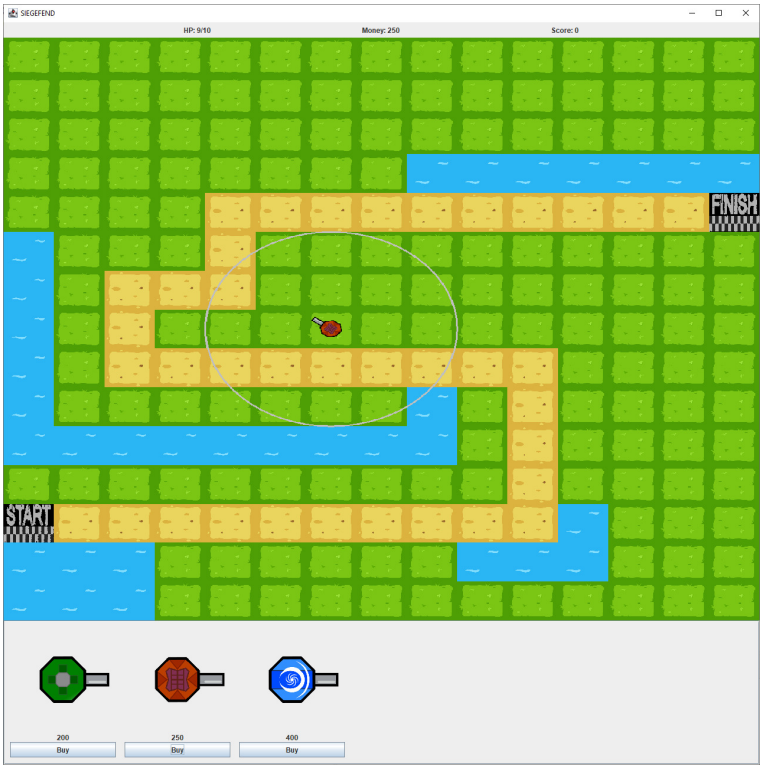


Figura A.4: Schermata raffigurante il range di una torretta